ISCO: A Practical Language for Logic-Based Construction of Heterogeneous Information Systems

Salvador Abreu
spa@di.uevora.pt

Departamento de Informática
Universidade de Évora
PORTUGAL

October 3, 2001

Abstract

Universidade de Évora's Integrated Information System (SIIUE) aims at representing the entire universe of concepts useful for the management and day-to-day operation of the Organization, as seen from the point of view of several different classes of users. It relies on ISCO, a logic programming language geared towards the development and maintenance of organizational information systems. ISCO is an evolution of our previous language DL and is based on a Constraint Logic Programming framework to define the schema, represent data, access heterogeneous data sources and perform arbitrary computations. This article presents the ISCO language and illustrates with a few examples how it may be applied to real-world situations stemming from diversified backgrounds. We claim that this approach pays off from a software project management point of view, and we compare it to a few other such initiatives.

Keywords: Logic in Databases, Deductive Databases, Knowledge Bases, Heterogenous Database Application Development.

1 Introduction

The relationship between Logic Programming and Databases has long been recognized as a very fruitful area of research, witness for example [13, 15, 21]. Accessing large amounts of loosely related information is one of the issues in data warehousing; some approaches taken to tackle this problem are discussed in [7, 8, 14]. A general approach consists in interposing a *mediator*[27] between the user and the data sources. The mediator is often formalized and implemented in a variant of a Logic Programming language such as Prolog. Many issues involved in using Logic to integrate different information repositories or to support schema evolution are discussed in the survey [12].

Universidade de Évora has developed and is currently deploying a general-purpose information system [1, 2], geared initially towards interacting with faculty members and staff but which aims to gradually fulfill the whole of the organization's internal information needs.

The approach chosen for SIIUE [1] can be summed up in the following design principles:

1. "Choose the right tool for each task". This is a very generic design principle, nevertheless it should be observed whenever constructing a new program development tool: if an adequate tool is already available, an attempt should be made to use it.

2. Use a Logic Programming basis for defining structure, data and procedures (declaratively). The LP dialect used for this purpose is called the *Information System COnstrution language* (ISCO), which is introduced in this article and is an evolution of the language DL presented in [1].

3. Structure the information in an object-oriented, single inheritance scheme. This approach allows for a good re-use of information and is quite intuitive in specifying complex class hierarchies as is inevitable in the case of the wide range of application domains for SIIUE.

4. Specify integrity constraints in the Logic language. Especially when associated with a form of Constraint Programming (such as constraints over finite domains), Logic Programming can be a very effective tool to specify a problem declaratively.

5. Ensure that all application-level interaction with the system is performed exclusively via the ISCO interface. Among other benefits, this enables using the ISCO language to specify access control mechanisms.

These choice has proven to be quite successful in solving a problem which is underspecified and of arguably

large scope, while having to endure severe restraints on the human resources available for its design and implementation. As the application had to be pushed into production state very early on, it is vital that the tools – particularly the language – incorporate mechanisms to support its evolution.

The development of a practical general-purpose tool that is able to access several heterogeneous information sources in a unified way, whilst uniformly representing both data and methods seems a very promising line of research: such is the challenge we are trying to meet in designing and implementing ISCO.

An initially unexpected goal in the design of ISCO and its environment, was to make it relatively easy for novice Prolog programmers to start using the language to construct useful applications.

Aside from the previously mentioned aspect of making a more uniform process of the definition and access to data, the tool introduced herein enables the construction of even more sophisticated mechanisms to deal with acknowledged software engineering issues occurring in real-world applications such as *schema evolution* [12] or *access control*: these and others may be undertaken entirely within the ISCO framework, partly by virtue of its building on top of Contextual Logic Programming [22] (see Section 8). In the case of access control, rules may be defined which may resort to the full expressiveness of Logic programs thereby subsuming in a major way the different situations that may occur, related to this issue.

The purpose of this article is to describe the tool central to the development of SIIUE: the ISCO language and its implementation, as well as the options and experimental observations which led to some of the design decisions that shaped the language. The architecture of the SIIUE application is also introduced and compared with other approaches to similar problems.

The article is structured as follows: after a brief introducion, the architecture of ISCO applications such as SIIUE is described in Section 2, in Section 3 the ISCO language is introduced and its features are discussed. Integrity constraints in ISCO are presented in Section 4, Section 5 deals with access control mechanisms and how they may be used to implement policies, in Section 6 the ISCO front-ends and back-ends are discussed, in Section 7 some comparisons are made with other work and finally Section 8 draws some conclusions and points to ongoing or future work on this software platform.

2 Architecture

The ISCO architecture is as follows: the primary interface to ISCO applications being the Web, there is a pool of web servers which run PHP [24] under Apache: this is designated as the *web layer*. A PHP module then allows the Apache processes to establish connections to Prolog processes, this layer is one of the *ISCO front-ends* (see Section 6). The Prolog processes form the *ISCO layer* proper and processes in this layer may, in turn, access several data sources which are designated as the *database layer* and is made up of *ISCO back-ends*.

The ISCO source program is compiled into the following forms:

1. A GNU Prolog executable with the native-code executable version of all ISCO predicates. This program implements the ISCO layer in an application.

2. A document describing the class hierarchy, with comments on every class' usage, on the attributes that are used as external keys, etc. This is especially useful in describing the actual system to non-technical persons. The format of the documentation can be either of HTML or LaTeX, the latter being used to produce high-quality cross-referenced PDF files.

The dialect of Prolog being used is planned to include a modularity extension enhancement: we are presently implementing a mechanism similar to that of Contextual Logic Programming [22], as it provides an elegant and effective framework in which to structure computations while retaining the simplicity of Prolog. In particular contextual definitions will allow for much cleaner (when compared to regular Prolog) means of:

- Defining higher-level procedures.
- Promoting code re-use.
- Supporting schema evolution.

The Prolog implementation being used, GNU Prolog [11], as an evolution of CLP(FD) also provides the constraint logic programming paradigm, which is a very useful extension to the traditional Prolog programming style in that it allows for problems to be solved by providing a-priori search-space pruning, through the constraint propagation mechanism. ISCO fully takes advantage of this feature.

3 ISCO: Language and Concepts

In an application, in order to describe classes and their behavior, a modeling language could have been used. Instead of resorting to an existing and evolving language such as UML [3] and its independent constraint specification language OCL [17], we opted for the design and implementation of a language –ISCO– based on well-understood concepts such as Constraint Logic Programming, a first-order logic description of classes and inheritance, class attributes, the values used to populate the classes and algorithms. The basis for ISCO

being Prolog, some issues in software engineering had to be addressed to make up for deficiencies of this language, when applied to typical information-system domains. The result is a revised logic programming language more suited to the reality of common database applications.

We feel that this approach provides more flexibility and dependability than would be feasible with more traditional tools, based on different formalisms.

An ISCO program is a regular Prolog program augmented with a syntax for describing classes, ISCO predicates and goals, sequences, integrity constraints and access control rules.

3.1 ISCO Predicates (Classes)

An ISCO predicate is similar to a Prolog predicate. However, its actual implementation may differ because some predicates may rely upon an external storage mechanism such as that provided by an RDBMS, which provides persistency for facts and the ability to efficiently process some queries. Syntactically, ISCO predicates (classes) are different from regular predicates in that:

- Arguments are explicitly named and typed.
- There may be constraints that specific arguments are automatically required to satisfy.
- Classes are organized hierarchically, forming an inheritance relation so that whatever is true for a member of a given class, will also be true for any member of any of its subclasses.
- There's a built-in notion of update for ISCO predicates, which may take on several forms (see Section 3.2).

The syntax for an ISCO program will be outlined with a few examples. A program is a sequence of ISCO definitions, interspersed with regular Prolog code (either clauses or directives).

An simple introductory example is given by the code fragment of figure 1, in which relations from two external databases (identified by the names `emp` and `ac` respectively) are used to construct a third relation, called `person`, which represents either students or employees.

This example illustrates a few features of the ISCO system, namely:

- The ability to integrate different data sources (`emp` could be stored in an Oracle database while `ac` could use a PostgreSQL database).
- The definition of computed classes, which can be seen as logic-based views.

Other features of ISCO will be described throughout the remainder of this article.

```
external(emp) class employee.
 id: sequence.
 name: text.

external(ac) class student.
 number: sequence.
 name: text.

computed class person.
 id.
 name: text.

rule :- employee(id=EID, name=NAME), ID=e(EID).
rule :- student(number=SID, name=NAME), ID=s(SID).
```

Figure 1: Simple ISCO Example

3.1.1 Programs and Definitions

An ISCO program is a sequence of *definitions*, *clauses* and *directives*. A definition may be one of:

- An *external* declaration which provides ISCO with the necessary information to access an external data source or sink, such as an ODBC-accessed database or an LDAP directory.
- A *class* definition, which is further discussed in Section 3.1.2.
- A *data* definition, in which specific values for a class may be provided.
- A *sequence* definition, which can be used to implement global counters in an application, which is described in Section 3.1.3.

Additionally, ISCO definitions may be interspersed with regular Prolog clauses and directives which are interpreted in the usual fashion, with the exception that goals are subject to the processing mentioned in Section 3.2.

3.1.2 Class definitions

A class definition is made up of three parts: the *head*, the *argument list* and the *body*. A class is introduced by its HEAD. It may be tagged with an optional sequence of class attributes which characterizes the class being defined. This is a sequence of comma-separated terms

Argument list: The argument list in a class declaration is is made up of a (possibly empty) sequence of argument definitions. The body of a class may be empty, in which case it will have no arguments (if it is a base class) or the arguments already present in its superclass (if it inherits from another class). An argument name is a Prolog "identifier" atom which uniquely identifies the argument being declared, within the scope of its class and all of its superclasses.

Argument declarations: Each argument declaration can come in one of two forms:

1. The first form is for regular arguments, which are explicitly typed at the time they are declared. Valid argument types are discussed in Section 3.4.

2. The second form is for arguments which are implicitly typed by constraining them to take values only in the set of values specified by some argument in another class. In this case the type is inferred to be the same as that of the target argument.

The attributes of an argument are a possibly empty sequence of Prolog terms. These describe attributes specific to the argument which immediately precedes them and generally represent constraints that the argument in question must satisfy for a tuple to be acceptable for the class.

Remainder of a class declaration: Classes may be suffixed with attributes similar to those which can appear as left arguments to the `class` keyword. This feature is simply an alternate syntax and does not change the semantics of a declaration.

Because their contents is fixed, static classes must have their declaration followed by their tuples. This form of declaration translates to Prolog "database" predicates, in which clauses have an empty body.

computed classes are expected to contain one or more *rules*, which come after the body of the class. These rules are regular Prolog clauses, with the following differences:

- The clause head is always "`rule :-`", with no arguments. There may be more than one rule for any given class.

- The clause body may access the implicit head variables, which are named after the arguments in the class definition, rewritten to be all in upper-case in order to comply with the Prolog notation for variables.

- ISCO predicate calls are subject to the preprocessing discussed in 3.2, namely the non-positional argument syntax is automatically translated to regular Prolog calls.

3.1.3 Sequence declaration

One of the SQL features that is most useful in building applications is the `sequence` construction. ISCO incorporates this concept with minimal impact on the language structure, and in two forms:

1. Sequences may be declared in a syntactic form similar to that of regular classes and accessed through a pair of predicates which perform the sequence operations (fetch or set next value and inquire about the current value.)

   The explicit sequence syntax is not recommended and only exists in order to better support existing databases.

2. The declaration of and accesses to a sequence may be altogether omitted, by introducing a class argument of type serial, in which case a uniquely named sequence is implicitly declared. In this case, the class argument in question also gets a default value which automatically increments the counter when inserting values into the class.

For example, the directive at the end of Figure 2 could first insert the tuple[1] (id=1, name='Universidade de Évora') into the organizational_unit class, and then insert the tuple (id=2, name='Informatics Department', parent=1) into the same class. Since variable `PID` is

```
class entity.
  id: serial. key.
  name: text.

class organizational_unit: entity.
  parent: entity.id.

:- organizational_unit :=
     (id=PID, name = 'University of Évora'),
   organizational_unit :=
     (name = 'Informatics Department',
      parent=PID).
```

Figure 2: sequence/serial example

unbound at the time of the first subgoal, it will become instantiated with the value which actually gets used when inserting the tuple: the next value for the automatic sequence associated with the `id` argument specified in the `entity` class.

3.2 Goal syntax

A program is made up of class and predicate definitions. Goals which occur in clause bodies can be categorized as regular Prolog goals or ISCO goals. The latter are intended to map to queries to the underlying database or other back-end engine and may be classified in the following categories:

- **Simple queries**. These correspond to interrogations and conform to the syntax:

$$\mathsf{NAME}_{rel}\ \underline{(}\ \mathsf{TUPLE}_{query}\ \underline{)}$$

[1] See Section 3.3 for a definition of "tuple".

Where NAME_{rel} is the name of an ISCO class. TUPLE_{query} is a constrained query-tuple (see Section 3.3 for details.) For example, the query:

```
20 #< P, P #< 30,
organizational_unit(parent=P, name=N)
```

This query is equivalent to the conjunction of constraints and goals:

```
20 #< P, P #< 30,
organizational_unit(_, N, P)
```

but does not require knowledge of the argument positions.[2] Its meaning could be "what are the names of the organizational units whose parent identifier lies in the interval `21..29`?"

Simple queries behave like regular Prolog goals, in that their arguments are either bound, free or constrained on input, and become all-ground on completion of the query. Simple queries are non-deterministic and may therefore produce several solutions upon backtracking.

Arguments may be suffixed with an *ordering operator*, which indicates whether and how the corresponding argument is to be ordered when producing solutions.

- **Insertion update queries**. Queries which add tuples to the relation use the syntax:

  $$\mathsf{NAME}_{rel}\ \underline{:=}\ \underline{(}\ \mathsf{TUPLE}_{new}\ \underline{)}$$

  Arguments which are omitted from the tuple are assigned the default value, should it exist. For example:

  ```
  :- organizational_unit :=
       (name='Computing Services')
  ```

  Insertion queries are deterministic and require all arguments to be either ground or omitted. It is an error to omit an argument for which there is no default.

  For arguments of *evaluable types* (see Section 3.4.2), their value is evaluated before being inserted.

- **Modification update queries**. These queries replace existing tuples with new values. The syntax is:

  $$\mathsf{NAME}_{rel}\ \underline{(}\ \mathsf{TUPLE}_{query}\ \underline{)}\ \underline{:=}\ \underline{(}\ \mathsf{TUPLE}_{new}\ \underline{)}$$

  An update query includes two tuples, which are interpreted respectively as the *selection* and the *modified* values. For example:

  ```
  organizational_unit(
    name='Academic Services') :=
     (parent=12)
  ```

  would cause the `organizational_unit` whose name is `Academic Services` to change its parent unit identifier to `12`.

  Modification queries are non-deterministic and, as opposed to an SQL `update` query, alter tuples one-at-a-time. In order to change all tuples that satisfy the selection constraints, the modification update query must have its search space exhausted, ie. it must be made to backtrack over all solutions. This approach has a reading more consistent with the usual Prolog operational semantics and allows for certain useful programming dialects to be used.

  Similarly to insertion update arguments, arguments of *evaluable types* occurring in the modified tuple are evaluated before being inserted.

- **Removal queries**. These queries remove existing tuples from a relation and may be expressed with the syntax:

  $$\mathsf{NAME}_{rel}\ \underline{(}\ \mathsf{TUPLE}_{query}\ \underline{)}\ \underline{:\backslash}$$

  For example, the query:

  ```
  ID #> 1000, entity(id=ID) :\
  ```

  Removes all tuples for the `entity` class (and all its subclasses) for which the `id` argument takes values greater than 10000.

  Removal queries are non-deterministic in the same way as modification update queries: the tuples are deleted one-at-a-time.

3.3 Tuples

An ISCO tuple is the collection of arguments passed to an ISCO predicate used to form an ISCO goal. The arguments to the query may come in two forms:

1. **Prolog form**: it must have exactly the same number of arguments as those in the class declaration. This is also referred to as the *positional argument* syntax.

2. **Non-positional form**: arguments are specified as a sequence of terms of the form:

   $$\mathsf{NAME}_{arg}\ \underline{=}\ \mathsf{TERM}$$

   Where NAME_{arg} is the name of an argument as declared in the relation and TERM is the term associated with that argument.

[2] In fact, the non-positional query is compiled into the regular Prolog goal syntax.

There are two sorts of terms: *constraint/query* terms and *new value* terms. The former is used to specify constraints on an argument or to retrieve its value, while the latter is used exclusively to specify a new (ground) value for an argument.

Constraint/query terms may be:

- *Unbound*, in which case the variable will become bound to the resulting value, after the query is performed.
- *Ground*, in which case the corresponding value will be used as a constraint for the query.
- Not ground but *subject to FD constraints*, in which case the constraints involving the term will be compiled into a form acceptable to the back-end. There are implementation-defined restrictions on the kinds of constraints that may be used in this case: in the present version, only constraints involving the variable and a constant are permissible.

New value terms may be:

- *Ground*, in which case the term's value (after being evaluated) will be used for the corresponding argument.
- *Unbound*: the corresponding argument's default value will be used.

3.4 Data Types

Data types in ISCO can be classified in two categories:

1. Simple types that map to the back-end types. These include integer numbers, floating point numbers, booleans, text,[3], date/time[4] and serial. For the RDBMS back-ends, the serial type results in a unique sequence being created. It is possible to reference an existing serial class argument in order to share the counter.
2. Compound types which constitute a re-use of *class* definitions as a data type. At present, this is simply a convenience feature as there is no type reference operator: compound types may be thought of simply as macros.

3.4.1 Example

In order to illustrate compound types and sequences, an example may be helpful. Suppose we want to represent a hierarchy of locations as well as a *temperature* value, expressed in either K, C or F degrees; the following declarations could be used:

```
abstract class temperature.
  value: float.
  degrees: [k, c, f].
```

This definition for `temperature` could subsequently be used in other places. One example could be:

```
abstract class location.
  id: serial.
  latitude: float.
  longitude: float.

class city: location.
  name: text.
  minimum_temperature: temperature.
  maximum_temperature: temperature.
```

This example highlights two features of ISCO: the use of declared classes as data types and the automatic sequence creation. The `id: serial.` declaration states that class `location` (and all its subclasses) have an argument called `id` which, when omitted in an insertion operation (see Section 3.2), gets a default value which results from incrementing the associated counter.

3.4.2 Evaluable types

For some types, and in the new-tuple situation, argument values are evaluated before being passed on to the back-end: this is the situation with all numeric types.

For instance, the following modification update query based on the previous examples illustrates the issue:

```
class classroom: location.
  building: location.id.
  capacity: int.

:- classroom(building=123, capacity=C) :=
     (capacity=C+10).
```

Executing this query would increase the capacity of all classrooms in building `123` by 10 seats.

4 Integrity Constraints

One aspect of ISCO that is essential to its use as a practical tool is the ability to specify flexible integrity constraints. The approach we took consists in associating integrity constraints with class arguments or individual classes as well as cater for global constraints, within the scope of the entire application.

Global integrity constraints are specified as queries that must either be always true or always false, and which involve one or more ISCO predicates.

Informally, an integrity constraint is a goal which is evaluated whenever any of its dependencies change: this may come as a result of a change to a relation occurring directly in the goal, or indirectly, for instance if the constraint invokes a computed relation.

[3]The "`text`" ISCO data type corresponds, in terms of RDBMS back-ends, to all variations of the "`character`" type. Prolog-wise, the values are atoms.

[4]The ISCO "`date`" type is able to represent dates and times as a compound term and maps to the appropriate back-end type.

Integrity constraints look like clauses for the `true/0` and `false/0` predicates. For example the global integrity constraint:

```
false :-
  setof(COURSE,
        enrolled(_STUDENT, COURSE, _YEAR),
        COURSES),
  length(COURSES, L), L > 8.
```

states that no student may be enrolled in more than 8 courses in any given year.

This particular constraint would be evaluated whenever a change to the `enrolled/3` relation[5] occurs. Should the constraint become true – meaning that it had been violated – the update goal that triggered the situation would be canceled. There is presently no provision for other forms of knowledge base consistency maintenance, such as forms of database revisions.

5 Access Control

One of the objectives in defining and implementing a language such as ISCO was to equalize the various back-ends' capabilities with respect to access controls: for instance, PostgreSQL databases provide quite different protection mechanisms from those supplied in Oracle or, even more so, from non-RDBMS systems such as LDAP. Some of these back-ends provide no access control mechanisms whatsoever. Having to implement an access control policy for a large heterogeneous database application may prove very hard if not for the availability of a unifying higher-level specification mechanism.

Access controls mechanisms may assume many forms, but essentially may be reduced to checking whether an *agent* may perform an *operation* on a *class* or *class instance*. This general approach enables the definition of many different access control policies such as simple Unix-like user classifications, or more sophisticated ones such as role-based access control as in [19, 20]. Moreover, operations may be made to depend on the data itself, rather than just the class.

Syntactically, access control rules appear in the epilogue of a class definition (see Section 3.1.2) and look like Prolog clauses whose success or failure indicates the permissibility of the operations the rule refers to.

As is the case for computed class rules, access control rules may implicitly use the declared class arguments as all-uppercase variants of the argument names. Another parameter is required to implement access controls: the *performing agent*, which has the special variable name AGENT and may occur in the body of the access control rule.

Access control rules come in two flavors: *class* and *instance* rules. The former dictates what operations may be initiated on an ISCO predicate, while the latter acts as a filter on the data which determines whether the intended operation is permissible for any given tuple. These rules may be thought of as pre- and post-conditions on the query, as they are checked, respectively, before and after the data access is performed.

Access control rules behave hierachically, following the inheritance relation of the class declarations: access to a subclass is, by default, subject to the same rules as its superclass.

[5] Or any dependency thereof, should it be a computed relation.

6 Back-Ends and Front-Ends

Several services are amenable to being integrated into the ISCO framework: these include not only relational databases (directly or via ODBC) but also directory services such as LDAP, as well as other network services such as SNMP or DNS. At present the following stores have been implemented:

- ODBC data sources, through the UnixODBC interface.
- Direct PostgreSQL data sources. These may also be accessed via ODBC, but a more efficient interface was deemed useful.
- LDAP directories.

At the time of this writing, ISCO provides two different front-ends:

- A "stand-alone" Prolog front-end which may be used as a regular Prolog top-level augmented with the ISCO language features.
- A PHP front-end which is used to construct web-based applications. This interacts with the ISCO process pool following a straightforward protocol, which accounts for control and pure data (HTML) connections. The PHP front-end includes a version of the PiLLoW [6] library which may be used to easily generate HTML or XML, from within the ISCO application.

7 Related Work

On the applications side, another system whose development was prompted by motivations similar to those underlying SIIUE – and therefore the ISCO language – is SIFEUP[9, 26], the general-purpose information system built at Universidade do Porto's Engineering Faculty. The tools used in SIFEUP are more traditional, and do not rely on a Logic Programming core.

Several systems integrate a Logic Programming language with Relational Databases, a prominent one being Infomaster [14] which caters to some of the issues addressed by ISCO, and relies on an extended Logic

Programming engine, fitted with an abduction mechanism, to access heterogeneous databases. ISCO provides a different approach in that it targets other types of information sources such as networked directory services and its intended use is also distinct.

8 Conclusions and Future Work

SIIUE [1, 2] has already proven useful by permitting a number of practical applications to be developed on relatively short notice and with restricted human resources. The combination of an RDBMS with a Logic Programming core was confirmed to be useful in that it allows for complex computations to be performed on the information contained in the database.

While this article doesn't present a finished system, it does introduce a line of work which is currently being pursued at Universidade de Évora which builds upon the experience gained while developing the SIIUE framework and its first applications. Various other on-going research and development projects at Universidade de Évora are related to this work, these include:

- Natural language interface.

  One of the most challenging issues when constructing large database applications in general and decision support systems in particular, is the ability to automatically generate useful queries from a specification created by a non-technical user. Even for a technically savvy person, it is sometimes useful to be able to query the system in natural language.

  We are currently working towards an application which will provide support for queries specified in a simplified natural language (initially we will be targeting Portuguese, obviously), with concepts and vocabulary appropriate for the information contained within SIIUE.

- Use of Contextual Logic Programming.

  The case for Contextual Logic Programming [22, 5] (CxLP for short) has already been made from several points of view, see for example [4, 10, 16, 18]. It is our belief that using CxLP as a basic feature in ISCO should bring about tangible benefits to the language, from a practical application development point of view. Furthermore, features such as schema evolution may be expressed very naturally in CxLP. Work towards a CxLP implementation for GNU Prolog has started, and shall be reported on.

- Visual Programming language for ISCO.

  With grounds similar to those that motivate a natural language interface, to which we can add the desire to ease the definition process for the concepts which underlie SIIUE (eg. the class hierarchy), SIIUE may prove a fertile ground on which to experiment with visual programming. Work is presently underway to explore this line of research, namely through a Java/Prolog interface which will, among other things, provide the necessary graphical operations.

- Georeferenced Information.

  If the description of the locations in a module of an application such as SIIUE is combined with geographically referenced information, several of the tasks performed by the system can benefit from an enhanced user interface. This is especially true if we add to that the Natural Language interface already mentioned.

Acknowledgements

The authors would like to thank Luís Arriaga da Cunha, Vitor Nogueira, Luís Quintano and Gonçalo Marrafa for various discussions pertaining to the work described herein. An acknowledgement is also due to the administration of Universidade de Évora for its support in the development of the SIIUE project, in which the present work is integrated.

References

[1] Salvador Pinto Abreu. A Logic-based Information System. In Pontelli and Santos-Costa [25]. 1, 2, 8

[2] Salvador Pinto Abreu and Joaquim Godinho. Logic-based Network Configuration and Management. In *The 7th International Congress of European University Information Systems*, Berlin, March 2001. Humboldt University. 1, 8

[3] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison Wesley, December 1997. ISBN: 0-201-57168-4. 3

[4] M. Bugliesi. A declarative view of inheritance in logic programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 113–127, Washington, USA, 1992. The MIT Press. 8

[5] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *The Journal of Logic Programming*, 19 & 20:443–502, May 1994. 8

[6] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/CIAO Library for INTERNET/WWW Programming. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings Of the 1st Workshop on Logic Programming Tools for Internet Applications, JICSLP-96*, pages 43–62, 1996. 6

[7] Surajit Chaudhuri and Umeshwar Dayal. Data warehousing and olap for decision support (tutorial). In Peckham [23], pages 507–508. 1

[8] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997. 1

[9] Gabriel David and Lígia Ribeiro. Impact of the Information System on the Pedagogical Process. In *The 7th International Congress of European University Information Systems*, Berlin, March 2001. Humboldt University. 7

[10] E. Denti, A. Natali, A. Omicini, and F. Zanichelli. Robot control systems as contextual logic programs. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 12, pages 343–379. Elsevier, 1995. 8

[11] Daniel Diaz and Philippe Codognet. GNU Prolog: Beyond Compiling to C. In Pontelli and Santos-Costa [25]. 2

[12] Yannis Dimopoulos and Antonis Kakas. Information Integration and Computational Logic. CoRR arXiv: cs. AI/ 0106025, June 2001. 1, 1

[13] H. Gallaire and J. Minker, editors. *Logic and Databases*. Plenum, New York, 1978. 1

[14] Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: An information integration system. In Peckham [23], pages 539–542. 1, 7

[15] J. Grant and J. Minker. The Impact of Logic Programming on Databases. *Communications of the ACM*, 35(3):66–81, 1992. 1

[16] Jean-Marie Jacquet and Luís Monteiro. Communicating clauses: Towards synchronous communication in contextual logic programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP-92)*, pages 98–112, Cambridge, November 9–13 1992. MIT Press. 8

[17] Anneke Kleppe, Jos Warmer, and Steve Cook. Informal formality? the Object Constraint Language and its application in the UML metamodel. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 127–136, 1998. 3

[18] E. Lamma and P. Mello. Processing abductive reasoning via contextual logic programming. *Lecture Notes in Computer Science*, 567:336–??, 1991. 8

[19] B. Lerner and A. Habermann. Beyond schema evolution to database reorganisation. *ACM SIGPLAN Notices*, 25(10):67–76, 1990. 5

[20] L. Liu. Maintaining Database consistency in the Presence of Schema Evolution. In Robert Meersman and Leo Mark, editors, *Proceedings of the Sixth IFIP TC-2 Working Conference on Data Semantics (DS-6)*, Stone Mountain, Atlanta, May-June 1995. Chapman & Hall, London. 5

[21] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos, 1988. 1

[22] Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993. 1, 2, 8

[23] Joan Peckham, editor. *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. ACM Press, 1997. 7, 14

[24] The PHP Hypertext Processor. `http://www.php.net/`. 2

[25] Enrico Pontelli and Vitor Santos-Costa, editors. *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, number 1753 in Lecture Notes in Computer Science, Boston, MA, USA, January 2000. Springer-Verlag. 1, 11

[26] L. Ribeiro, G. David, A. Azevedo, and J.C. Marques dos Santos. Developing an Information System at the Engineering Faculty of Porto University. In *Proceedings of the European Coorperation in Higher Education Information Systems – EUNIS97*, Grenoble, France, September 1997. 7

[27] G. Wiederhold. Mediation to deal with heterogeneous data sources. *Lecture Notes in Computer Science*, 1580:1–??, 1999. 1