Modeling Role-Based Access Control in ISCO

Salvador Abreu(∗)

(∗) Departamento de Informática, Universidade de Évora and CENTRIA (FCT/UNL)
Portugal
`spa@di.uevora.pt`

Abstract

In this article we describe the mechanisms offered by the ISCO [2, 1] language for defining and implementing access control and homologation, within the scope of a large-scale information system. ISCO is a Logic-based system which integrates multiple heterogenous information sources such as classical relational databases or LDAP directories.

The *rule-based* access control mechanisms present in ISCO can be used to implement a wide variety of access control policies, ranging from simple owner/group/other permissions to role-based access control with capabilities which extend further, such as content filtering, as we shall demonstrate.

We claim that the access control mechanisms that can be implemented using ISCO subsume existing ones in expressiveness. An example of usage within a real-world application is also provided.

Keywords: Information Systems, Logic Programming, Access-Control Mechanisms, Deductive Databases, Web Interfaces.

1 Introduction

The relationship between Logic Programming and Databases has long been recognized as a very fruitful area of research, witness for example [12, 14, 17]. 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 [6, 7, 13]. A general approach consists in interposing a *mediator*[21, 22] 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 [11].

Universidade de Évora has developed and is currently deploying a general-purpose information system framework [1, 3], geared initially towards interacting with faculty members and staff but which aims to gradually fulfill the whole of the organization's internal information needs. This system is being developed around ISCO, a language based on Prolog which relies on external data sources/sinks to store and retrieve Datalog information.

Systems such as Infomaster [13] have been designed and implemented which allow access to multiple heterogenous information sources, unified with a Logic Programming framework. ISCO [2, 1, 3] goes beyond what such systems propose, in that it not only allows for uniformly accessing an existing set of information sources, but also proposes a framework where the information structure may evolve and be extended.

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. This came as a result of the declarations in ISCO, which can largely correspond to an object-relational data model but which can be incrementally extended to encompass computations, treating these with a uniform syntax and semantics.

Aside from the previously mentioned aspect of making a more uniform process of the definition and access to data, the tools introduced in ISCO enable the construction of even more sophisticated mechanisms to deal with acknowledged software engineering issues occurring in real-world applications such as *schema evolution* [11] 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 [18], which greatly eases software development by structuring the Prolog namespace for predicates.

In the case of access control, rules may be defined which may resort to the expressiveness of Logic programs thereby unifying and enhancing in major ways the different offerings related to permissions and access control.

The purpose of this article is twofold: (1) to describe the mechanisms within ISCO which specify and enforce different forms of access control, and (2) To report on the usage of the supplied programming mechanisms to specify and implement an application-related feature: information validation and homologation.

The remainder of this article is structured as follows: the most salient features of the language as well as the architecture of ISCO applications such as SIIUE are succinctly outlined in section 2. Section 3 addresses the issues of identifying and authenticating an agent. Section 4 discusses how access control policies may be described and programmed within ISCO. In section 5 the implementation of ISCO rules and the integration of access control are discussed. Section 6 describes a form whereby the concepts of information validation, homologation and delegation may be implemented. Finally, in section 7 we compare our approach with others from the literature, comment on the current state of the implementation and point at directions for future work.

2 Application Development with ISCO

A information system application built with ISCO can be described as being made up of the following components:

- A web interface layer, which may be simply accessed by direct users of the system. Applications in this layer are mostly developed using a mix of HTML and PHP. Ideally, this layer should provide only minimal content, as most HTML or XML can be conveniently generated below.
- A computational layer, in which information from the database layer is unified and where processes and computations may take place. The programming language used here is ISCO. Offline applications may be developed at this level.

  Interface procedures that are intended to be re-used frequently should be designed in this layer, as they may make use of the PiLLoW [5] library to conveniently generate HTML or XML from within the ISCO program.
- A heterogeneous storage layer, usually implemented by relational databases, which provides persistency.

The ISCO language described in [2] builds on the basis of Prolog, adding data description features which may be used to access persistent storage such as that provided by RDBMSs. ISCO descriptions may be programmed from scratch or be generated from the schema of an existing database.

The foremost example of an ISCO application is Universidade de Évora's Integrated Information System (SIIUE), which is being actively used to provide a uniform interface to on-campus information. SIIUE aims at dealing with every aspect of academic life, ranging from registrations and student grades to online information on research projects and publications or detailed course information. Earlier versions of some parts of SIIUE have been described in [1, 3].

One benefit of ISCO is the relative ease with which the structure of an information system may be inspected: a good demonstration is the natural language query interface which is being developed for SIIUE, described in [20].

3 Identification and Authentication

The ISCO architecture can be summarized by figure 1: the different layers correspond to actual physically different networks, interfacing each pair of layers which have contact.

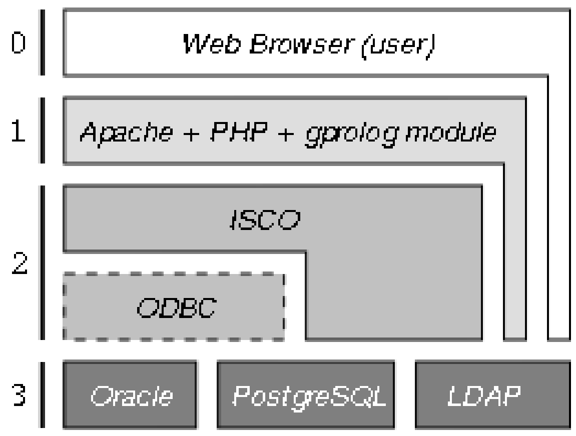

Figure 1: Layers in ISCO

The physical separation is provided to ensure that access to higher-numbered layers is exclusively performed by hosts on the layer immediately below.

There is the requirement that, since all validation and authorization is performed by the ISCO layer, the layers above only access any application data via ISCO.

Beyond the aforementioned security issues, there is another practical reason for this layout, having to do with the main ISCO application (SIIUE, see [1]) and the form in which it is implemented: a lot of code had to be rewritten when ISCO became available. Insulating the Web server/PHP layer (1) from the database layer (3) helped ensure that no old code –direct database accesses in particular– would execute unwittingly.

- User agents are found in layer 0 and communicate exclusively with the HTTP servers (in layer 1) and the LDAP directory servers, which may be thought of as lying simultaneously in layers 1 and 3.
- The web servers in layer 1 only do very basic operations, such as verifying the identity of the user and ensuring a secure channel through to layer 0 over which the session may take place. They mey perform cosmetic work such as embedding the output of the layer 2 servers in uniform-looking pages. These are implemented as an Apache server with PHP, supplemented with a module for communicating with a Prolog engine.
- Layer 2 are the ISCO processors proper: these run Prolog images which implement a front-end appropriate for communicating with the PHP/Prolog interface [9] developed at Universidade de Évora. The Prolog (ISCO, actually) programs running in layer 2 have the physical possibility of accessing the multiple information sources that make up the information system, which lie in layer 3. Databases in layer 3 may be accessed either through an ODBC interface[1] or, in the case of PostgreSQL, directly via a native interface. LDAP directories are also accessed transparently as these get mapped to ISCO classes, similarly to what happens with relational databases. The mechanisms used to access LDAP directories are described in [19].

[1]This was also the object of a development effort a Universidade de Évora: an UnixODBC interface for GNU Prolog.

Based on this organization, the process for a regular interaction with any ISCO application goes on as follows:

1. The user (layer 0) establishes a secure session (SSL or TLS) to the service through a web browser.

2. The interface (in layer 1) cooperates with the browser and an LDAP server to identify and authenticate the user. This information is subsequently made available for the duration of the session.

3. The application in layers 1 and 2 starts executing. It is made up of HTML, PHP and ISCO components. The last two components may use the information pertaining to the session, in particular the user's identity.

4 Access Control Rules

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 a *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 [15, 16]. Moreover, operations may be made to depend on the data itself, rather than just the class.

Access Control Rules (ACRs) constitute the basic mechanism for specifying authorization in ISCO. These come in two flavours: class and instance-based. The rationale behind such a distinction is probably what distinguishes ISCO from, say, what is possible with regular RDBMSs: the ability to allow or disallow an operation on the basis of the data it would operate on itself, rather than simply what relations it involves.

ACRs are expressed as Prolog clauses, which complement the regular integrity constraints already present within the ISCO language. The reader may wish to read [2] for a discussion on integrity constraints in ISCO.

4.1 Structure of ACRs

Access Control Rules need to specify three things:

1. A *subject* (or *agent*). The subject is transported in the environment, and may be referred to via the predicate agent/1, which unifies its argument with the identity of the agent wanting to perform the operation. ACR clauses may refer to the agent via the implicit variable AGENT, which is available in the entire ACR (head and body).

2. An *operation*. The operation is a designator for ISCO operations on class predicates: class access (empty), read, insert, modify and delete.

3. An *object*. The object of an ISCO class operation is a class name, for an instance operation it's the instance itself. The distinction between class and instance Access Control Rules is made through the *operation*, as previously described: only the class access operation does without an instance.

ACRs must immediately follow the class declaration which they will apply to. Should a class have no direct successful[2] applicable ACR, it will inherit the applicable rules from its superclass. A root class with no ACRs behaves as if it had a single ACR allowing all operations, indiscriminately.

4.2 Class-Based ACRs

A class ACR is the simplest form of access control rule found in ISCO. It basically states that a given agent is allowed to enquire about the metadata associated with a given class. Satisfaction of the class ACR is a prerequisite to the execution of any goal pertaining to that class, ie. class ACRs are always evaluated, prior to instance ACRs.

The syntax for class ACRs is as follows:

access :- BODY

Where BODY is an ISCO Prolog goal, which may make use of the implicit variable AGENT to identify the subject performing the inquiry.

```
abstract class confidential.

access :- is_member(id=AGENT, unit=management).
access :- is_privileged(AGENT).
```

Figure 2: Example of a Class-Based ACR

For example, consider the ISCO class definition of figure 2: in this case, an empty abstract class "confidential" is being defined, likely with the goal of using it as a superclass whenever convenient. As a consequence of this declaration, all subclasses of `confidential` that do not explicitly introduce their own class access control rules, will only be visible and accessible for agents which satisfy the goal "`access`" when applied to `confidential`. In this example, this means that the AGENT is either privileged or a member of the `management` unit.

[2]By "successful" it is meant that it should have a clause that succeeds. If that ACR is to be definitive –i.e. override its inherited behavior– it should discard all potential alternatives, with a cut.

In order for a specific action to be performed on a class, the agent must first pass the class access control rules for that particular class. Note that inheritance is applicable, allowing for class access specifications to be written simply for the topmost relevant classes.

4.3 Instance-Based ACRs

Similarly to what happens in class ACRs, instance-based access control rules are attached to individual class definitions but their goal is to allow or disallow the completion of instance-specific operations, namely the read, insert, modify and delete actions.

Opposite to class ACRs, however, is that instance-based access control rules will apply to individual tuples: the outcome of the rule evaluation can be thought of as a post-condition whereas the class ACR would be a pre-condition for the application of the intended action.

The syntax for instance ACRs is: $\boxed{\text{MODE } \underline{\text{access}} \ \underline{\texttt{:-}} \text{ BODY}}$ where MODE indicates the intended operation and may be one of:

- read – this instance ACR will apply to the ISCO goal $\boxed{\text{RELATION}(\text{ARG}_{\text{list}})}$ and stands for regular read-only queries to the information system. The semantics is similar to that of a Prolog goal call.

- insert – this instance ACR will apply to the ISCO goal $\boxed{\text{RELATION } \underline{\texttt{:=}} \ (\text{ARG}_{\text{list}})}$ and stands for the insertion of new tuples into a relation. This operation has a semantics close to Prolog's `assertz/3`.

- modify – this ACR applies to the ISCO goal $\boxed{\text{RELATION}(\text{ARG}_{\text{list}}) \ \underline{\texttt{:=}} \ (\text{ARG}_{\text{list}})}$ and stands for the modification of existing tuples in a relation.

- delete – this instance ACR applies to the ISCO goal $\boxed{\text{RELATION}(\text{ARG}_{\text{list}}) \ \underline{\texttt{:\textbackslash}}}$ and stands for the removal of tuples from a relation. The semantics of this operation is close to Prolog's `retract/1`.

BODY is an ISCO Prolog goal. As is also the case in class ACRs, the body may make use of the implicit variable AGENT to identify the agent performing the query.

The instance-based ACRs are what really puts ISCO apart from other approaches, in that they provide a mechanism whereby access to information may be controlled on the basis of the information itself, not simply its classification (ie. its class).

Consider the simplified example of figure 3, inspired from the application being developed for the Academic Services, which involves the ability of a particular student to enroll in a specific course, for a given semester. First, a few comments which should help in specifying the problem:

- The *ability to enroll* can be thought of as the ability to insert, modify or delete tuples from the `enrolled` class.

- A student should only be able to enroll himself, not another student.

- Any professor or T.A. should be able to find out which students are enrolled in any course they teach – but not to alter any of these.

```
mutable class enrolled.
  student: student.id.        % variable STUDENT
  course: course.id.          % variable COURSE
  year: int.                  % variable YEAR
  semester: [1, 2].           % variable SEMESTER

read access :-
  teaches(AGENT, COURSE, YEAR, SEMESTER), !.
_ANY access :- AGENT = STUDENT.
```

Figure 3: Example of an Instance-Based ACR

Notice that, also for the example of figure 3, rules pertaining to the eligibility of a student to enroll in a specific instance of a given course are *left out*. Examples of such rules include the maximum total number of credits per semester, whether the prerequisites of the course have been met by the student, whether the course is permissible for the graduation program the student is registered for, etc.

The issue here is that the enforcement of such constraints can be guaranteed through the use of another ISCO language feature: global integrity constraints. Figure 4 shows a possible implementation using global integrity constraints. In this example, it is assumed that a computed relation can_enroll/4 has been defined, which includes only those tuples for which it is true that a given student may enroll in a specific course.

```
% -- restrict maximum credits per semester -----------
% -- triggered on changes to 'enrolled' and 'course' -
false :-
     bagof(CR, C^Y^S^
       ( enrolled(ST, C, Y, S),
         course(id=C, credits=CR)), CRs),
     sum_list(CRs, TOTAL_CREDITS),
     TOTAL_CREDITS > 20.

% -- restrict on course prerequisites ----------------
% -- triggered on changes to 'enrolled' --------------
false :-
     enrolled(STUDENT, C, YEAR, SEMESTER),
     \+ can_enroll(STUDENT, C, YEAR, SEMESTER).
```

Figure 4: Gloabl Integrity Constraints for Figure 3

It is also possible to replace some of the functionality of global integrity constraints with judiciously designed access control rules. Continuing with the previous example, and considering that students may never alter their enrollment status, the ACRs could be rewritten as per figure 5. Notice, in this figure, the

```
read access :- teaches(AGENT, COURSE,
                       YEAR, SEMESTER), !.
read access :- AGENT = STUDENT.

insert access :-
    AGENT = STUDENT,
    !,
    course(id=COURSE, credits=CREDITS),
    total_credits(STUDENT, YEAR,
                  SEMESTER, TOTAL),
    NEW_TOTAL is TOTAL+CREDITS,
    NEW_TOTAL <= 20.

modify access :- !, fail.
delete access :- !, fail.
```

Figure 5: Replacing Integrity Constraints with Instance ACRs

ACRs for modify and delete access which are set to always fail: this in effect captures the notion that this relation may only be modified monotonically. In fact, these two rules could have been entirely omitted if the class had not been declared mutable (see figure 3): in that situation, the predicates that implement the update and delete operators would simply not have been generated, thereby preventing the operations from taking place at all.

Besides the fact that integrity constraints require a transaction in order to operate, there are other efficiency advantages to using instance ACRs to implement integrity constraints, as their scope is more limited and their implementation (see the next section) has a lower computational overhead.

To summarize, global integrity constraints should only be used in situations where operations that perform updates to more than one relation are involved: in that case, only one integrity constraint needs to be specified while, with instance ACRs, several cases would have to be described.

5 ISCO Clauses and ACR Implementation

Access Control Rules are implemented by the ISCO compiler, integrated with the generation for the various access modes to class predicates. Briefly, the compiler produces between one and four distinct predicates[3], one for each of the allowed access modes. The actual number of predicates depends on the class attributes: a static class will only have the read predicates, a regular class will have both the read and insert predicates whereas a mutable class will have all four predicates defined.

These predicates are made up of regular Prolog clauses with positional arguments and a name which hashes both the class name and the operation. The actual number of arguments is given by:

[3]In fact, there are more than four, but these suffice for the purpose of discussing the ACR implementation.

- the number of arguments in the class (local and inherited),
- internal arguments that describe the back-end connection and actual argument usage masks (this is done in order to manage the database back-end more efficiently) and
- possibly another set of arguments that reflect the new tuple, for the modify operation.

5.1 ISCO Goals and Clauses

Before describing how ACRs are actually incorportated into the compiled form of ISCO access clauses, a brief description of the structure of these clauses is useful. Consider the case of the modify code, i.e. the clause that would support the execution of an ISCO goal such as that shown in figure 6, which would

```
enrolled(year=2001, course=20, student=S) :=
    (semester=2)
```

Figure 6: Example for modify operations

set course number 20 in schoolyear 2001/02 to occur in the second semester, for every student S. This goal will behave nondeterministically, in that it will bind successive values of S, one for each student satisfying the selection criteria.

Assuming that relation `enrolled` is being mapped to an SQL[4] back-end, the supporting clause will have the structure shown in figure 7. A brief analysis of the structure of this clause

```
isco_modify_enrolled(CONN,
                     S,C,Y,M,
                     SS,CC,YY,MM, MASK) :-
  ..., % (1) generate SQL select, returning OID
  ..., % (2) invoke SQL select (nondeterminate)
  ..., % (3) generate SQL update, using OID
  .... % (4) invoke SQL update.
```

Figure 7: Prolog code for ISCO modify operation

is as follows:

- The clause head carries extra information which includes a reference to the back-end connection (the `CONN` argument) and a mask describing what arguments are actually being used by the calling goal (the `MASK` argument). Moreover, there are two clause head arguments for each class argument: one for the selection part and one for the modification part; these are the `S,C,Y,M` and `SS,CC,YY,MM` arguments respectively.
- Part `(1)` of the generated Prolog code creates an SQL `select` query which, in this case, will look something like "`select oid, student from enrolled`

[4]The compiler has knowledge about different RDBMS back-ends, and is able to generate differentiated SQL code for each of them: most notable is the ability of certain DBMSs to handle inheritance, such is the case with PostgreSQL.

`where year=2001 and semester=2`". The purpose of this section of the clause is to uniquely identify the tuple which matched the conditions specified by the bindings of the selection variables (the `SELARGS` head variables): in order for a selection variable to be included in the `where` SQL clause, it will have either to be ground or be a constrained variable, in the sense of CLP(FD).

The SQL output variables in the `select` statement always include the `oid` as well as any variables that have been specified in the output mask (the last head argument).

Note that the exact SQL statement being built varies with the state of the selection variables and the output mask.

- Part `(2)` of the clause executes the SQL statement and fetches the relevant output variables. In this case, head variable `S` and local clause variable `OID` are bound. This part is nondeterminate, i.e. it may backtrack to produce more solutions.

  Note: should Prolog execution remove the choice point in any way other than by exhaustion of the alternatives, for example by executing a cut, a special "choicepoint destructor" function is invoked which releases all external resources allocated within the choice point. This ensures that there will be no memory leaks due to the nondeterminate interface to, for instance, SQL back-ends, while avoiding the transfer of the entire set of solutions.

- Parts `(3)` and `(4)` use the values returned by part `(2)` in order to construct and execute an SQL statement of the form "`update enrolled set semester=2 where oid=OID`". The execution of this statement is determinate, as there is only one tuple with `oid=OID`.

  Notice that this SQL statement cannot be generated ahead of time, as it may alter variables which get bound as a result of executing parts `(1)` and `(2)` of the clause body.

5.2 Incorporating ACRs into ISCO Clauses

The inclusion of access control rules into ISCO clauses is rather straightforward. Consider the clauses for each of the ISCO operations; adding ACRs will, after performing some bookkeeping tasks, cause the sequence of actions to look like the following:

1. Evaluate the class access control rule. Should this test fail, the entire operation will fail. In a sense, the class ACR behaves as a precondition for the operation to be performed.

2. Perform the non-modifying part of the intended operation. This part is only empty in the insert case and corresponds to parts `(1)` and `(2)` of figure 7.

3. Evaluate the instance access control rule, as applied to the tuple returned by step 2.

   Should the instance ACR be of the form `AGENT = ARG`, where `ARG` is a class member name, it will be evaluated before step 2, leading to a more optimized query in the sense that it will only produce solutions which satisfy the ACR.

   This approach is similar to the Andorra Principle [4], also known as "sidetracking", which consists in reordering goals so as to evaluate the determinate ones first.

4. Should there be a second tuple specified in the operation, as is the case for the modify operation, the instance ACR is evaluated again, before the modifying part of the clause, this time with the values from the new tuple.

5. If the instance access control rule(s) succeeded, the modifying part of the intended operation is performed. This part is empty for read operations.

When compared with ISCO code generation without access control rules, this process has three extra steps (1, 3 and 4), which are fairly straightforward to insert into the generated clause bodies.

6 Homologation and Delegation

From the point of view of an organization's administration, it is not sufficient to express validity of the information found in a database simply as "`Joe is a salesman of Foo-type widgets`": other questions arise, such as "***Says who?***" or "***On what authority?***"

Whoever is ultimately responsible for the information contained in the information system (the company's CIO, for instance) needs to be able to rely on the accuracy and timeliness of the data, as well as to delegate on others the ability to input, alter or validate information. Furthermore, it is also important to pinpoint the process that led to some piece of information being present.

6.0.1 Definitions

The reliability of this process can be ensured with two concepts, *Homologation* and *Delegation*, which are defined by the Collaborative International Dictionary of English, version 0.44 of May 2001, as:

Homologation: "Confirmation or ratification (as of something otherwise null and void), by a court or a grantor."

Delegation: "The act or process of authorizing subordinates to make certain decisions."

6.0.2 Implementation

These concepts need not be a *feature* of the ISCO language: they seem to be useful for the construction of a wide range of applications –namely all which have to model workflow– but the inclusion at the language level is not necessary. We now proceed to illustrate how to use the ACR mechanism and other ISCO language features, in order to effectively implement homologation and delegation.

6.0.3 Validity Status

A piece of information may have a *validity status*, which can be represented via the ISCO class declaration of figure 8. The

```
mutable class validity.
  id:     serial. % the unique sequence number
  object: homologated.id. indexed. % refers to?
  valid:  bool.               % is it valid?
  reason: validity_state.id.  % why (in)valid?
  when:   datetime.           % when?
  who:    entity.id.          % who says so?
  note:   text.               % a comment.
```

Figure 8: Validity Status class definition

purpose of most arguments is self-explanatory. There may be more than one `validity` entry for each object, but only the highest-numbered (as per the `id` argument) is considered.

6.0.4 Using Validity Information in ACRs

Access control rules for classes subject to the homologation process may be implemented with an empty common superclass, `homologated`, for which an instance ACR (recall section 4.3) may be specified that will ensure:

- That agents with the statutory right to modify the information are granted modification access (insert, modify and delete).
- That read access is only granted in case the information is *valid* or the agent has explicit access to it.

Figure 9 illustrates a possible implementation. The first in-

```
abstract class homologated.
  id: serial.

      access. % class ACR
    _ access :- owner(CLASS, ID, AGENT), !.
read access :- validity(object=ID, valid=V,
                        id=_ <@) -> V=true.
```

Figure 9: ACRs to Enforce Homologation

stance ACR grants all forms of access to the owner of the datum. The second instance ACR, which only specifies read access, succeeds if the most recent `validity` entry for the intended object has a `valid` value of `true`. Note the <@ operator applied to the `id` argument, which ensures that solutions are sorted in by decreasing `id`: the first solution will therefore be the one with the highest value for `id`. Effectively, this rule shields information which is in an intermediate state from being accessed at all, by unauthorized agents.

An ACR such as this one implements the management rule "*rather show no information at all than incorrect one*."

As soon as the homologation process carries on to create a new `validity` record for the same object, with the valid argument set to `true`, read queries to subclasses of `homologated` will succeed, regardless of who the agent is. This is true because the new record's `id` argument will have a higher numeric value and will therefore come first in the specified solution sort order.

The homologation classes and procedures may be enriched in several ways, namely to cover workflow-related issues, such as deadlines or completion notification.

6.0.5 Delegation

Implementing delegation is really quite straightforward: all that needs to be done is to supplement the instance ACRs of figure 9 with a clause that translates to the concept "whoever has the right to perform an operation may delegate it onto someone else". A possible realization of a simple "delegate all rights" policy can be achieved by replacing the clause first instance ACR by those of figure 10. Note that for this particular imple-

```
_ access :- owner(CLASS, ID, AGENT), !.
_ access :- owner(CLASS, ID, O),
            delegation(CLASS, O, AGENT), !.
```

Figure 10: Delegation ACRs

mentation to work, there must be a class `delegation` which indicates the active delegations. Note also that the delegations themselves may be subject to homologation.

7 Conclusions and Further Work

The proposed mechanism suits the needs it aimed to fulfill quite nicely. As this work is part of a larger effort, namely the design and implementation of Universidade de Évora's Integrated Information System (SIIUE), and most directions of future development –both for ISCO and in particular for the Access Control mechanisms– will be guided by the experience we gain from actual usage.

Our experience in porting the existing DL [1] code base to the new framework is proceeding well, notwithstanding the unexpected issue with some of the developers being somewhat aversive of programming in Prolog.

When compared with RDBMS-centered systems, ISCO clearly provides a much more flexible and powerful mechanism for access control. In terms of the development process, the inheritance mechanism allows for a relatively high rate of code reuse, including the case of access control. Other efforts such as Yang and Kifer's FLORA [23] focus more on the ability of the system to provide a more general Logic Programming language.

The developments which we plan to work on next include:

- Better SQL code generation, possibly by delaying the execution of database query goals, which should allow for a closer to optimal construction of queries. At present, SQL back-end queries are treated individually.

- Further integration of the CLP aspects of GNU Prolog [10, 8]. This could include the addition of suitable constraint domains, such as strings.
- Re-implementation of some aspects both of the tools (namely the ISCO compiler) and the resulting applications to make use of the Contextual Logic Programming implementation which is becoming usable in GNU Prolog.

Acknowledgements

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

References

[1] Salvador Abreu. A Logic-based Information System. In Enrico Pontelli and Vitor Santos-Costa, editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, volume 1753 of *Lecture Notes in Computer Science*, pages 141–153, Boston, MA, USA, January 2000. Springer-Verlag.

[2] Salvador Abreu. Isco: A practical language for heterogeneous information system construction. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. INAP.

[3] Salvador Abreu and Joaquim Godinho. Logic-based Network Configuration and Management. In Jan Knop, editor, *The Changing Universities - The Role of Technology – The 7th International Conference of European University Information Systems*, Lecture Notes in Informatics, Berlin, March 2001. German Informatics Society (GI). ISBN 3-88579-339-3.

[4] Salvador Abreu, Luís Moniz Pereira, and Philippe Codognet. Improving backward execution in the andorra family of languages. In *Proceedings of the International Joint Conference and Symposium on Logic Programming*. MIT Press, 1992.

[5] Daniel Cabeza and Manuel Hermenegildo. Distributed WWW programming using (Ciao-)Prolog and the PiLLoW library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.

[6] Surajit Chaudhuri and Umeshwar Dayal. Data warehousing and olap for decision support (tutorial). In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 507–508. ACM Press, 1997.

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

[8] Philippe Codognet and Daniel Diaz. Compiling Constraint in `clp(FD)`. *Journal of Logic Programming*, 27(3), June 1996.

[9] João Conceição and Salvador Abreu. Interfacing Prolog and PHP. Submitted for publication, July 2001.

[10] Daniel Diaz and Philippe Codognet. GNU Prolog: Beyond Compiling to C. In *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*.

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

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

[13] Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: An information integration system. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 539–542. ACM Press, 1997.

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

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

[16] 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.

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

[18] 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.

[19] Pedro Patinho and Salvador Abreu. ProLDAP - Api em Prolog para Directórios LDAP. In *IV Conferência de Redes de Computadores*. FCCN, November 2001. (in Portuguese).

[20] Luis Quintano, Irene Rodrigues, and Salvador Abreu. Relational information retrieval through natural lanaguage analysis. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. INAP.

[21] G. Wiederhold. Mediators in the Architecture of Future Information System. *IEEE Computer*, 25(3), 1992.

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

[23] Guizhen Yang and Michael Kifer. FLORA: Implementing an Efficient DOOD System Using a Tabling Logic Engine. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 1078–1093. Springer, 2000.