Efficient and Flexible Access Control via Logic Program Specialisation*

Steve Barker
Dept. of Computer Science
King's College
The Strand, WC2R 2LS, UK
steve@dcs.kcl.ac.uk

Michael Leuschel, Mauricio Varea
School of Electronics and Computer Science
University of Southampton
Highfield, SO17 1BJ, UK
{mal,mv}@ecs.soton.ac.uk

ABSTRACT

We describe the use of a flexible meta-interpreter for performing access control checks on deductive databases. The meta-program is implemented in Prolog and takes as input a database and an access policy specification. We then proceed to specialise the meta-program for a given access policy and intensional database by using the LOGEN partial evaluation system. In addition to describing the programs involved in our approach, we give a number of performance measures for our implementation of an access control checker, and we discuss the implications of using this approach for access control on deductive databases. In particular, we show that by using our approach we get flexible access control with virtually zero overhead.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*security, integrity, and protection*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*partial evaluation*; D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence**]: Automatic Programming; D.1.6 [**Programming Techniques**]: Logic Programming; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*logic programming*

General Terms

Security, Performance, Languages

Keywords

Database Access Control, Datalog, Partial Deduction, Cogen Approach, Program Transformation

*Work partially supported by European Framework 5 Project ASAP (IST-2001-38059).

PEPM'04, August 24–26, 2004, Verona, Italy.

1. INTRODUCTION

The issue of controlling a user's ability to exercise access privileges (e.g., read, write, execute privileges) on a system's resources has long been an important issue in Computer Science. In recent years, there has been considerable interest in access control in research and in practice. All aspects of security have been given particular prominence with the advent of the Web. In a number of surveys, security issues have been reported by enterprises as being of paramount concern when deciding policies on the publication of Web data, and the availability of Web resources (see, for example, [6]). Security issues, including access control issues, will be of particular importance in the emerging Semantic Web, and for e-commerce applications (see, for example, [11]).

In recent years, a number of researchers have developed some sophisticated access control models in which access control requirements may be expressed by using rules that are employed to reason about authorised forms of access (see, for example, [12], [5], and [3]). In these approaches, access to resources are expressed by using rules that define the conditions that must be satisfied in order for a permission/denial/authorisation to hold. Expressing access control policies by using rules is natural, and enables many implicit permissions/denials/authorisations to be expressed in a succinct manner. However, an important practical issue that arises with the rule-based approach to access control is the problem of efficiently evaluating access requests when access control requirements are implicitly specified. The problem of efficiently evaluating access requests with respect to rule-based specifications of access policies has become especially important recently, as organizations require increasingly complex forms of access control policies for protecting resources. These complex forms of policy specifications are potentially expensive to compute.

For each of the approaches described in [12], [5], and [3], proposals are made for attempting to ensure that access requests are evaluated efficiently when access control requirements are specified implicitly. In [12] and [5], *view materialisation* approaches are described for attempting to optimize access control checks. The motivation for the view materialisation approach is to make explicit the access control information that is implicitly defined in rule form. Making explicit the implicitly specified access control information, means that access requests can be evaluated by considering explicitly recorded facts rather than these facts having to be derived at query evaluation time. Unfortunately, view materialisation is not so efficient to use when large num-

bers of *parametric derivation rules* [4] are used to express access control requirements and when the specification of access control requirements changes dynamically e.g., when *user session information* [3] is used in the course of deciding whether an access request is authorised.

Rather than using view materialisation techniques, the approach described in [3] enables access requests to be efficiently evaluated by utilizing *constraint logic programming* techniques [19]. The approach described in [3] makes use of specialised constraint solvers, rather than view materialisation techniques, for the efficient evaluation of access requests in situations where large numbers of parametric derivation rules (e.g., rules that express temporal constraints on user access) would be expensive to compute, and when changes to an access policy are performed dynamically as a consequence of a user's session management. Nevertheless, the potential optimization of access requests by using program specialisation techniques is not considered in [3]. Furthermore, each of the approaches described in [12], [5], and [3] assumes that access control is expressed with respect to coarse-grained data objects (e.g., files and directories), and that an answer to an access request on a data item is simply whether access is allowed or not. In contrast, the work in [1] has the significant computational attraction of exploiting request modification techniques to combine the decision on allowing access with the actual generation of authorised data that may be released to answer a user's access request. However, the approach described in [1] does not exploit specific access request optimization methods.

In contrast to [12], [5], and [3], we describe an approach to the problem of access request evaluation where large numbers of parametric derivation rules are required in order to specify access policy requirements; where fine-grained access to data items is required (e.g., access to atomic formulae); where the answer to a user access request generates the set of logical consequences that the user is permitted to see; and where access control information is to be exploited for performance gains.

In overview, we describe an access control checker that is implemented by using a meta-program that is written as a logic program. The meta-program takes as input an access control program and a database. The approach that we describe enables the access control program to be specialised, in order to reduce the amount of run-time information that needs to be considered when deciding whether an access request is authorised. In effect, the approach ensures that a minimal amount of information is considered at access request evaluation time. Specifically, the user session information that applies at the time of an access request is used with a form of access control program that is specialised by using the relatively static information explicitly specified in the access control program.

In practice, the rules defining an access control policy are not subject to frequent changes. As such, this relatively static information may be exploited for program specialisation. Moreover, an access control request is a request that is made by a specific (authenticated) user to perform a specific operation (i.e., read, write, execute, etc.) on a specific database item. Exploiting the information about a user's identity and the access privilege the user wishes to exercise on a database object can be exploited to specialise a program for access control and hence can be exploited for computational advantage.

Although meta-interpreters have previously been developed for efficient constraint checking on databases [17], to the best of our knowledge, no approach has yet been proposed in the literature for generating specialised access requests via a meta-interpreter that manipulates access requests, access control policies and databases as object level expressions, and that precompiles access checking for certain access requests. In this paper, we describe a technique to obtain a specialised access control checker that is more efficient to use than using a database and access control program directly because some of the propagation, simplification and evaluation process is precompiled.

We consider the use of *role-based access control (RBAC)* policies [3] for specifying authorised forms of access to database objects. In RBAC, the most fundamental notion is that of a role. A role is defined in terms of a job function in an organization (e.g., a *doctor* role in a medical environment), and users and access privileges on objects are assigned to roles. Moreover, access privileges on objects (i.e., *permissions*) are assigned to roles (e.g., a doctor has the permission to change a patient's prescriptions). RBAC policies have a number of well documented attractions [23], and are widely used in practice [9]. Although we restrict our attention to RBAC policies in this paper, it should be noted that RBAC is a more general form of access control model than the *discretionary access control* and *mandatory access control* approaches that predate RBAC [8], and the approach that we describe can be used with more powerful access control methods than RBAC (e.g., the *status-based access control* model [2]). It follows that our approach is widely applicable.

We represent an RBAC policy by using a logic program. The use of logic programs for representing access control policies has been recognised in a number of recent works (see, for example, [12] and [3]). Logic programs enable access policies to be expressed by using high-level declarative languages for which formally well defined semantics and operational methods with attractive theoretical properties (e.g., termination) are known to exist.

The rest of this paper is organized as follows. In Section 2, some basic notions in logic programming and the LOGEN partial evaluation system (which has been improved since the system used in [17]) are briefly described. In Section 3, we briefly describe an RBAC model, and the formulation of RBAC policies by using logic programs. In Section 4 we describe the metaprogram that we use for the evaluation of access requests on databases with respect to a formulation of an RBAC policy. In Section 5, we present some performance measures for an implementation of our approach, and we discuss the results that we report. Finally, in Section 6, some conclusions are drawn and suggestions for further work are made.

2. PRELIMINARIES

In this section, we briefly describe some syntactic and semantic issues relating to logic programming. We then give a brief overview of partial evaluation using the LOGEN system.

2.1 Syntax and Semantics

The *RBAC* model and the *RBAC* policies that we describe in later sections are expressed in the language of (function-free) normal clause form logic ($Datalog^{\neg}$), with certain predicates in the alphabet Σ of the language having

a fixed intended interpretation. As we only admit function-free clauses, the only terms of relevance to Σ will be constants and variables. Hereafter, we denote variables that appear in clauses by using symbols that appear in upper case (at least the first character), and constants will be denoted by lower case symbols.

A normal clause is a formula of the form:

$$C \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n \qquad (m \geq 0, n \geq 0).$$

The *head*, C, of the clause above is a single *atom*. The *body* of the clause (i.e., $A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n$) is a conjunction of literals. Each A_i literal ($i \in \{1, .., m\}$) is a *positive literal*; each $\neg B_j$ literal ($j \in \{1, .., n\}$) is a *negative literal*. In the case of a negative literal, the relevant type of negation is *negation as failure* [7]. A clause with an empty body is an *assertion* or a *fact*. A clause with a non-empty head and a non-empty body is a *rule*. A *relational database* is a set of facts; a normal *deductive database* is a set of normal clauses. The set of facts in a deductive database Δ is referred to as the *extensional* part of Δ, (the EDB of Δ), and the set of rules in Δ is referred to as the *intensional* part of Δ (the IDB of Δ).

In our representation of a database, a fact of the form $p(c_1, \ldots, c_n)$ (where each subscripted p is an arbitrary n-place predicate and $c_i, \forall i \in \{1, \ldots, n\}$, are constants) is represented as an atom of the following form:

$$fact(p(c_1, \ldots, c_n)).$$

A clause of the following form (where each subscripted p is an arbitrary n-place predicate and each subscripted t is a term)

$$p_1(t_1, \ldots, t_n) \leftarrow p_2(t_i, \ldots, t_j), \ldots, p_m(t_k, \ldots, t_l).$$

is represented in our databases by using an atom of the following form:

$$rule(p_1(t_1, \ldots, t_n), [p_2(t_i, \ldots, t_j), \ldots, p_m(t_k, \ldots, t_l)]).$$

The access control programs that we consider are always *locally stratified* (a realistic assumption for most practical policies) and hence have a unique *perfect model* [20]. Having a 2-valued model theoretic semantics is important for ensuring that authorised forms of access are unambiguously specified.

2.2 Partial Evaluation and the LOGEN System

Partial evaluation [14] is a source-to-source program transformation technique that specialises programs by fixing part of the input of some source program P and then pre-computing those parts of P that only depend on the known part of the input. The so-obtained transformed programs are less general than the original, but can be much more efficient. The part of the input that is fixed is referred to as the *static* input, while the remainder of the input is called the *dynamic* input.

Partial evaluation is especially useful when applied to interpreters. In that setting, the static input is typically the object program being interpreted, while the actual call to the object program is dynamic. Partial evaluation can then produce a more efficient, specialised version of the interpreter, which is sometimes akin to a compiled version of the object program [10].

The LOGEN system [16] is a so-called *offline* partial evaluator for Prolog, i.e., specialisation is divided into two phases, as depicted in Figure 1:

- First a *binding-time analysis* (BTA for short) is performed which, given a program and an approximation of the input available for specialisation, approximates all values within the program and generates annotations that steer (or control) the specialisation process.
- A (simplified) *specialisation phase*, which is guided by the result of the BTA.

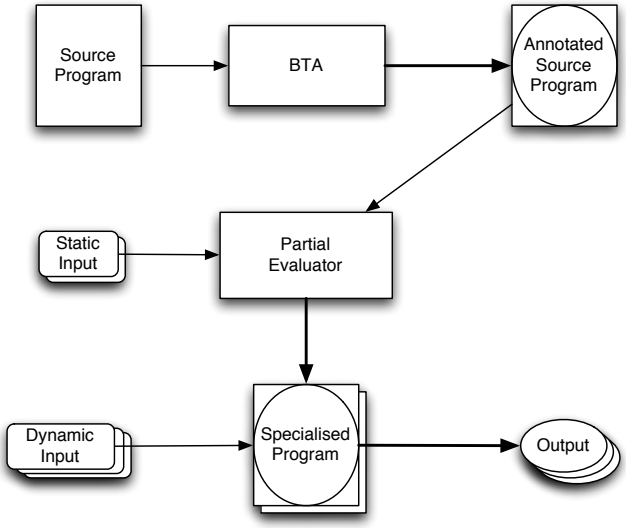

Figure 1: Offline Partial Evaluation

Because of the preliminary BTA, the specialisation process itself can be performed very efficiently, with predictable results. Also, as shown in [15], the LOGEN system is well suited to specialise interpreters, something that we will aim to exploit in our approach.

3. RBAC POLICIES AS LOGIC PROGRAMS

In this section, we describe a simple type of RBAC policy that may be used to protect the information in databases. More specifically, the one type of policy that we describe here is based on the $RBAC^P_{H2A}$ model that is formally defined in [3]. We only consider one type of access control policy in this paper because our principal concern is to describe the generalities of using a meta-programming approach for access request checking, access policy program specialisation by LOGEN, and performance evaluation. It should be noted, however, that any of the policies from [3] may be represented by using our meta-programming approach to access control checking, with minor modifications.

We call an access control program that is defined in terms of the $RBAC^P_{H2A}$ model, an $RBAC^P_{H2A}$ *program*. This type of program is a finite set of normal clauses specified with respect to a domain of discourse that includes:

- A set $\mathcal{U}$ of *users*.
- A set $\mathcal{O}$ of *objects*.

- A set $\mathcal{A}$ of *access privileges.*
- A set $\mathcal{R}$ of *roles.*

In an $RBAC^P_{H2A}$ program, a user is specified as being assigned to a role by using definitions of a 2-place *ura* predicate, and the assignment of an access privilege on an object to a role is expressed by using definitions of a 3-place *pra* predicate in the $RBAC^P_{H2A}$ program. The semantics of these predicates in an arbitrary $RBAC^P_{H2A}$ program Π may be expressed thus:

- $\Pi \models ura(u, r)$ iff user $u \in \mathcal{U}$ is assigned to role $r \in \mathcal{R}$;
- $\Pi \models pra(a, o, r)$ iff the access privilege $a \in \mathcal{A}$ on object $o \in \mathcal{O}$ is assigned to the role $r \in \mathcal{R}$.

By separating the assignment of users to roles from the assignment of permissions to roles it is possible for user-role and permission-role assignments to be changed independently of each other in implementations of $RBAC^P_{H2A}$ policies. Thus, access policy maintenance is simplified (relative to the discretionary access control policies that were, until recently, used as a matter of course to help to protect the information in databases).

In the $RBAC^P_{H2A}$ model, specified in [3], an $RBAC^P_{H2A}$ *role hierarchy* is defined as a (partially) ordered set of roles. The ordering relation is a role seniority relation. In an $RBAC^P_{H2A}$ program Π, a 2-place predicate $senior\_to(r_i, r_j)$ is used to define the seniority ordering between pairs of roles. That is, the role $r_i \in \mathcal{R}$ is a more senior role (or more powerful role) than role $r_j \in \mathcal{R}$. If r_i is senior to r_j then any user assigned to the role r_i has at least the permissions that users assigned to rule r_j have. More formally, the semantics of the *senior_to* relation may be expressed thus:

- $\Pi \models senior\_to(r_i, r_j)$ iff the role $r_i \in \mathcal{R}$ is senior to the role $r_j \in \mathcal{R}$ in an $RBAC^P_{H2A}$ role hierarchy.

The *senior_to* relation may be defined as the reflexive-transitive closure of an irreflexive-intransitive binary relation *ds*. The semantics of *ds* may be expressed, in terms of an $RBAC^P_{H2A}$ program Π, thus:

- $\Pi \models ds(r_i, r_j)$ iff the role $r_i \in \mathcal{R}$ is senior to the role $r_j \in \mathcal{R}$ in an $RBAC^P_{H2A}$ role hierarchy defined in Π and $\neg\exists r_k \in \mathcal{R}\ [ds(r_k, r_j)\ \wedge\ ds(r_i, r_k)]$ where $r_k \neq r_i$ and $r_k \neq r_j$.

Assuming the lattice of role hierarchies to be complete, an $RBAC^P_{H2A}$ role hierarchy is defined by the following set of clauses (in which '_' is an anonymous variable):

$$
\begin{array}{lcl}
senior\_to(R1, R1) & \leftarrow & ds(R1, \_). \\
senior\_to(R1, R1) & \leftarrow & ds(\_, R1). \\
senior\_to(R1, R2) & \leftarrow & ds(R1, R2). \\
senior\_to(R1, R2) & \leftarrow & ds(R1, R3), senior\_to(R3, R2).
\end{array}
$$

In RBAC models generally, senior roles are assumed to inherit the access privileges on objects that are assigned to junior roles in an $RBAC^P_{H2A}$ role hierarchy. An $RBAC^P_{H2A}$ role hierarchy enables many authorisations to be implicitly defined, thus simplifying the expression of access control policies.

In RBAC, users activate and deactivate roles in the course of *session management* [3] as required to perform the tasks associated with a job function. In [3], the notion of a user $u_i \in \mathcal{U}$ activating a role $r_j \in \mathcal{R}$ in a session is represented by using a set of rules of the following form:

$$active(U, R) \quad \leftarrow \quad activate(U, R), C.$$

In this context, a user u_i requests to be active in a role r_j by appending an $activate(u_i, r_j)$ fact to an $RBAC^P_{H2A}$ program via a GUI. An $active(u_i, r_j)$ fact will be implicitly appended to an $RBAC^P_{H2A}$ program whenever u_i has requested to be active in a role r_j and the set of conditions C, on u_i's activation of the role r_j is satisfied. Any *activate* assertion that enables the user u_i to be active in role r_j may be retracted by u_i when u_i no longer wishes to be active in r_j, and all of the *activate* assertions for a user are automatically retracted when the user logs off of the system.

An *authorisations clause* [3] is used to define that a user $u_i \in \mathcal{U}$ has the $a_k \in \mathcal{A}$ access privilege on object $o_l \in \mathcal{O}$. In the case of $RBAC^P_{H2A}$ programs, the authorisations clause is defined thus:

$$
\begin{array}{lcl}
permitted(U, A, O) & \leftarrow & ura(U, R1), \\
 & & active(U, R1), \\
 & & senior\_to(R1, R2), \\
 & & pra(A, O, R2).
\end{array}
$$

The rule that defines *permitted* is used to express that a user U may exercise the A access privilege on object O if: U is assigned to the role $R1$, U is active in $R1$, $R1$ is senior to a role $R2$ in an $RBAC^P_{H2A}$ role hierarchy, and $R2$ has been assigned the A access privilege on O.

In the context of specialising an $RBAC^P_{H2A}$ program Π, we note that the definitions of *ura*, *pra*, *senior_to* and *ds* are part of the object level information that is used to protect the object level database in our approach. Moreover, the sets of clauses defining the extensions of the *ura*, *pra*, *ds* and *senior_to* relations are static relative to the set of *active* atoms that are implicit in Π. That is, the set of *active* facts will change dynamically as users activate and deactivate roles. The aim of our approach is to specialise $RBAC^P_{H2A}$ programs to enable efficient access control checks to be performed by only considering user session information expressed via the set of *active* facts that is current at the time of a user's access control request.

4. THE META-INTERPRETER

In this section, we describe the meta-interpreter that we propose for efficient access request evaluation on deductive databases that are protected by $RBAC^P_{H2A}$ programs. We restrict our attention to a consideration of read access.

The following Prolog code is part of the meta-interpreter that is used to execute the $RBAC^P_{H2A}$ programs that we have described for access control:

```
holds_read(User,not(Object)) :-
          \+(holds_read(User,Object)).
holds_read(_User,Object) :- built_in(Object).

holds_read(User,Object) :-
          permitted(User,read,Object),
          fact(Object), call(Object).

holds_read(User,Object) :-
          permitted(User,read,Object),
          rule(Object,Body),
          l_holds_read(User,Body).

l_holds_read(_U,[]).
```

```
l_holds_read(U,[H|T]) :- holds_read(U,H),
                         l_holds_read(U,T).

built_in('='(X,X)).
built_in('is'(X,Y)) :- X is Y.

holds(U,O) :- holds_read(U,O).
```

The full code can be found in Appendix B. The definition of `permitted` that is assumed in this example is that we described above for implementing an $RBAC^P_{H2A}$ policy, to wit:

```
permitted(User,Op,Obj) :- ura(User,Role),
                          active(User,Role),
                          senior_to(Role,R2),
                          pra(R2,Op,Obj).
```

This paper only considers the definition of authorisations by `permitted/3`, as its goal is to apply $RBAC^P_{H2A}$ policies. Any number of alternative definitions of `permitted` may be used to implement different access policies (see [3] for other definitions of authorisation clauses that may be used), and do not require any modifications to our meta-interpreter in order to process those access requests.

EXAMPLE 1. *Consider an* $RBAC^P_{H2A}$ *program* Π *with the following sets of facts:*

$$DS = \{ds(r1, r2)\}.$$

$$ACTIVE = \{active(u1, r1), active(u2, r2)\}.$$

$$URA = \{ura(u1, r1), ura(u1, r2), ura(u2, r2)\}.$$

$$PRA = \{pra(r1, read, s(\_)), pra(r2, read, p(\_)), pra(r2, read, q(\_, \_)), pra(r1, read, r(\_, \_))\}.$$

Moreover, suppose that Π *is used to protect the following database* Δ *in which* p *and* s *are EDB predicates and* p *and* q *are IDB predicates:*

$$fact(p(X)).$$
$$fact(s(X)).$$
$$rule(q(X, Y), [p(X), p(Y)]).$$
$$rule(r(X, Y), [q(X, Y), s(X)]).$$

The access request `holds_read(u1,q(A,B))` *by user* u1 *to read all instances of* q *from* Δ*, can be specialised by* LOGEN *into:*

```
holds_read(u1,q(A,B)) :- holds_read__0(A,B).

permitted__1(B,C) :- active(u1,r1).
permitted__1(D,E) :- active(u1,r2).

permitted__4(B) :- active(u1,r1).
permitted__4(C) :- active(u1,r2).

holds_read__3(B) :- permitted__4(B), p(B).

holds_read__0(B,C) :- permitted__1(B,C),
                      holds_read__3(B),
                      holds_read__3(C).
```

By inspection, it is possible to see that the effect of such a specialisation is to reduce a predicate like `permitted`, which is defined in terms of the relatively static predicates *ura*, *pra*, *ds* and *senior_to*, to tests on the run-time information that is generated in the course of session management, i.e., *active* facts.

5. PERFORMANCE MEASURES

In this section, we give some performance measures for the meta-programming approach that we propose for evaluating access requests on deductive databases that are protected by using an $RBAC^P_{H2A}$ program. Our testing involved comparing the evaluation of access requests on (i) a non-specialised, and (ii) a LOGEN specialised $RBAC^P_{H2A}$ meta-programs. For comparison's sake, we also measured versions of the $RBAC^P_{H2A}$ that have no access control. These versions are implemented directly as Prolog clauses and hence needed no meta-interpreter to run.

The $RBAC^P_{H2A}$ programs that we use in our tests have included a definition of the *senior_to* relation that represents an $RBAC^P_{H2A}$ role hierarchy with 53 roles arranged as a complete lattice, and with each node/role of outdegree 3 or indegree 3. The *senior_to* relation has been materialised into a set of 312 pairs of ground binary assertions[1].

We have experimented with variants of the $RBAC^P_{H2A}$ role hierarchy by increasing the depth of the role lattice. The summation that follows describes the number of pairs of roles in the *senior_to* relation as defined by the $RBAC^P_{H2A}$ role hierarchy that we use in testing:

$$N + 2 \sum_{i=1}^{d-1/2} 3^i + (N_d * P_{>1})$$

In the summation above, N is the total number of nodes in the role lattice, d is the depth of the lattice, N_d is the number of nodes at depth d in the lattice, and $P_{>1}$ is the number of paths of length 2 or greater from a node at depth d.

The unique bottom element in all of the $RBAC^P_{H2A}$ role hierarchies that we use in testing is assigned the read permission on all of the logical consequences of the databases that we use in testing. Moreover, our testing is based on a single user that is assigned to the most senior role/unique top element in the $RBAC^P_{H2A}$ role hierarchies/complete lattices that are used in our testing. Access requests are evaluated for this user. Our choice of user-role assignment and permission-role assignments imply that our tests are based on a worst-case scenario that involves the maximum amount of inheritance of permissions whenever an access request is evaluated.

The queries that we use in testing involve computing two binary relations *tcp* and *cycle*, and a unary relation q. The *tcp* relation is the transitive closure of a 2-place predicate p; the *cycle* relation involves computing a transitive closure in order to determine elements in the reflexive closure of p; the definition of q is a variant of the well-known *win* program.[2] The *tcp* program was chosen for inclusion in testing because of its practical significance; *cycle* was chosen because it involves some expensive recursive processing; the q program was chosen because it combines recursion and negation, and is a useful benchmark test for performance studies.

The definitions of the *tcp*, *cycle* and q predicates are expressed in our database thus:

[1]In this case, we use a partial materialisation approach such that only the role hierarchy is materialised (but not the authorisations).

[2]The *win* program describes a two-player game in which a player wins if his or her opponent has no move to make. The formalisation of this two-person game may be expressed by the clause: $win(X) \leftarrow move(X, Y), \neg win(Y)$.

$$tcp(X,Y) \leftarrow p(X,Y).$$
$$tcp(X,Y) \leftarrow p(X,Z), tcp(Z,Y).$$

$$cycle(X,Y) \leftarrow p(X,Y).$$
$$cycle(X,Y) \leftarrow cycle(X,Z), p(Z,Y).$$

$$q(X) \leftarrow p(X,Y), \neg q(Y).$$

The 2-place p predicate is defined by a set of 2495 facts. A total of 499 p facts are used to represent the chain:

$$p(a_1,a_2), p(a_2,a_3), \ldots, p(a_{498},a_{499}), p(a_{499},a_{500}).$$

An additional 1996 p facts are used to achieve a fan-out factor of 5 [21]. That is, for each p fact with the first argument a_i, where $1 \leqslant i \leqslant 499$, there are four p facts with the second argument of p equal to the value b_j, where $1 \leqslant j \leqslant 4$. For example, $p(a_1,b_1), p(a_1,b_2), p(a_1,b_3), p(a_1,b_4)$. For the *cycle* program, at the n^{th} call to *cycle*, a chain of $(n-1)$ elements in the transitive closure of p is computed, and hence the goal clause $p(a_n,a_{n-1})$ is evaluated. An additional fact $p(a_{500},a_1)$ is added to the 2495 p facts used with *tcp* to represent the end of the cycle.

The successful $tcp(a_1,a_{500})$ query that we use in our testing involves computing a 500 element chain starting from the element a_1 and ending with the element a_{500}. To evaluate the *tcp* query by using SLD-resolution, a search space comprising 499 SLD-trees with root $\leftarrow tcp(a_n,a_{500})$, where $1 \leqslant n \leqslant 499$, was generated. Each of these 499 SLD-trees spawns 5 subtrees; 4 of which fail, and one that succeeds. The four failing cases have a b_j value ($1 \leqslant j \leqslant 4$) as the second argument of a p fact; the succeeding subtree terminates with an answer clause of the form $p(a_s,a_t)$ where $t = s+1$, $1 \leqslant s \leqslant 499$ and $2 \leqslant t \leqslant 500$. The $cycle(a_1,a_1)$ query used involves computing every chain from a_1 to a_w ($2 \leqslant w \leqslant 500$) in the transitive closure of p, until $p(a_{500},a_1)$ succeeds and hence $p(a_1,a_1)$ succeeds. The failing query in our suite of tests ($tcp(a_1,a_{501})$) is an attempt to compute a 501 element chain that terminates at the element a_{501}. The successful $q(a_1)$ query involves generating 499 failing SLD-trees for the 499 evaluations of the $\neg q(c_m)$ subgoal, where $1 \leqslant m \leqslant 499$. The one successful SLD-derivation is generated from the ground clause: $q(a_1) \leftarrow r(a_1,c_{500}), \neg q(c_{500})$.

The results of the testing of our example queries are summarised in Table 1 (for the non-specialised case), and Tables 2 and 3 (for the specialised case). The queries denoted by Q_1, Q_2, Q_3 and Q_4 in these tables have the following meanings:

- Q_1 is the successful $tcp(a_1,a_{500})$ query run 10 times;
- Q_2 is the failed $tcp(a_1,a_{501})$ query run 5 times;
- Q_3 is the successful $cycle(X,Y)$ query (all 145,850 solutions);
- Q_4 is the successful $q(X)$ query (all 1,170 solutions).

The query times are expressed in seconds, and are usually averaged over several runs. The time needed to generate the compiler from the interpreter (i.e., performing the second Futamura projection [10]) was $0.040s$. The prior binding-time analysis was performed (once and for all) by hand using LOGEN's new graphical interface that allows easy annotation and provides colouring feedback on static and dynamic parts.[3] To achieve the good results it was essential to follow the approach from [15] (see also Appendix C). Timings were obtained on a Powerbook G4 1Ghz, 1GB SDRAM, with SICStus Prolog 3.11.0 and Mac OS X 10.3.2. Runtimes for XSB were obtained on the same machine using XSB Prolog 2.6. In our experiments, we make use of XSB's distinctive feature: it terminates for both recursive and non recursive datalog programs. This mechanism is known as *tabling* in XSB Prolog [22], and has been proved very useful in deductive databases. Tabling allows, for instance, the evaluation of query Q_3, which only XSB Prolog can run ensuring termination.

Query	*With* $RBAC^P_{H2A}$	*Without* $RBAC^P_{H2A}$	*Overhead*
Q_1 *(SICStus)*	*0.135 s*	*0.003 s*	*0.132 s*
Q_1 *(XSB)*	*0.100 s*	*0.000 s*	*0.100 s*
Q_2 *(SICStus)*	*1.372 s*	*0.004 s*	*1.368 s*
Q_2 *(XSB)*	*0.100 s*	*0.000 s*	*0.100 s*
Q_3 *(XSB)*	*1.460 s*	*1.080 s*	*0.380 s*
Q_4 *(SICStus)*	*9.64 s*	*0.060 s*	*9.580 s*
Q_4 *(XSB)*	*0.109 s*	*0.010 s*	*0.099 s*

Table 1: Average retrieval times for the non-specialised case.

Table 1 shows how much overhead is introduced by the access control policy. For example, query Q_3 that takes 1.08 seconds to retrieve information takes an extra 0.38 seconds when access control is performed. Ideally, we want to minimise the overhead introduced by the $RBAC^P_{H2A}$ policy. By specialisation of the meta-interpreter, we achieve a speedup that considerably reduces this overhead, as illustrated in Table 2. It can be observed that after applying the LOGEN tool, the average retrieval time is improved by a factor of up to 42. In all cases the retrieval time after specialisation falls between the average times of the two previous approaches, i.e., *with* and *without* access control.

Query	*Specialisation Time*	*Average Runtime*	*Speedup*
Q_1	*0.010 s*	*0.007 s*	*19.3*
Q_2	*0.010 s*	*0.032 s*	*42.88*
Q_4	*0.010 s*	*0.950 s*	*10.15*
Q'_1	*0.010 s*	*0.003 s*	*45*
Q'_2	*0.010 s*	*0.004 s*	*343*
Q'_4	*0.010 s*	*0.060 s*	*120.5*

Table 2: Retrieval times (running in SICStus) for the specialised case.

[3] An automatic binding-time analysis is in the final stages of implementation (as can be guessed from the screenshot in Appendix D) and it will hopefully be able to annotate our interpreter automatically. However, it is acceptable to perform the BTA by hand, as the annotation only has to be generated once and it is independent of the database as well as the access control policy.

There is, of course, a penalty introduced by this approach: the *specialisation time*, i.e., the time it takes *logen* to figure out a specialised version of the meta-interpreter with an $RBAC^P_{H2A}$ policy. However, Table 2 shows that adding together the *average runtime* and the *specialisation time* does not exceed the original times. By adjusting the annotations (i.e., marking more calls as unfoldable), a more aggressive specialisation can be obtained. This is shown in the second part of the table, where Q'_i is the same query as Q_i, but taking the *senior_to* clause into account as well (which is likely to remain unchanged for a long time).

Query	*Specialisation Time*	*Average Runtime*	*Speedup*
Q_1	*0.010 s*	*0.026 s*	*3.85*
Q_2	*0.010 s*	*0.024 s*	*4.17*
Q_3	*0.010 s*	*1.220 s*	*1.20*
Q_4	*0.010 s*	*0.016 s*	*6.81*
Q'_1	*0.010 s*	*0.010 s*	*10*
Q'_2	*0.010 s*	*0.008 s*	*12.5*
Q'_3	*0.010 s*	*1.130 s*	*1.29*
Q'_4	*0.010 s*	*0.013 s*	*8.38*

Table 3: Retrieval times (running in XSB) for the specialised case.

Table 3 shows how the previous results compares when the Prolog Engine includes tabling. It can be observed that for the more aggressive criteria the time is reduced and even reaches, in most cases, the ideal without-access-control figure aimed (i.e., overhead=0). For query Q_3 the specialised interpreter (see Appendix A) is actually almost identical to the database without access control.[4] We have thus actually achieve what is called "Jones optimality" [13, 14, 18, 15] (called the "optimality criterion" in [14]). The only drawback of the aggressive specialisation is that each time *senior_to* changes there is an overhead of $10ms$ for specialisation (as well as the time needed to load the new specialised interpreter, which was around $10ms$ in our experiments). Since this clause does not change very often, we are not paying a high price in terms of access control flexibility.

Our testing is based on the scenario where a user inherits all access privileges on all objects (i.e., logical consequences) from the most junior role in the $RBAC^P_{H2A}$ role hierarchy. In practice, access request evaluation will involve significantly less permission inheritance, and user access to data objects will be far more constrained than we have considered in our testing. We consider only this scenario because access control requirements will be highly application specific. The more specific access control restrictions that will apply in practice will enable LOGEN to specialise access control programs to a much greater extent than we have considered, and therefore more impressive speedup can be expected in practical applications.

6. CONCLUSIONS AND FURTHER WORK

We have described a partial-deduction approach for specialising access control checking on deductive databases. Our approach uses the LOGEN system to specialise $RBAC^P_{H2A}$ programs. The results of our experiments using the LOGEN system, have revealed that program specialisation produces significant improvements in access request evaluation times on deductive databases protected by $RBAC^P_{H2A}$ programs. In fact, by using access control information for specialisation, it is possible to evaluate access requests on deductive databases as efficiently as evaluating the same requests without processing access control information (in case the user is actually allowed to access the information). Hence, our approach makes it possible to incorporate access control checks into access request evaluations on deductive databases without incurring any overheads.

There are a number of additional issues to investigate in the context of optimizing access requests on policy information in P2P and B2B applications. It would also be interesting to apply our approach to emerging access control models for controlling access to Web resources (see, for example, [2]). We intend to investigate these issues in future work.

[4]We believe the fact that our specialised interpreter runs slightly slower is probably due to caching issues.

7. ACKNOWLEDGEMENTS

The authors would like to thank Annie Liu and the anonymous referees for their very helpful comments, as well as Stephen-John Craig for helping with the experiments in Logen.

8. REFERENCES

[1] S. Barker. Protecting deductive databases from unauthorized retrieval and update requests. *Journal of Data and Knowledge Engineering*, 23(3):231–285, 2002.

[2] S. Barker. Web usage control in rsclp. In *Proc. 18th IFIP WG Conf. on Database Security*, 2004.

[3] S. Barker and P. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. on Information and System Security*, 6(4):501–546, 2003.

[4] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM TODS*, 23(3):231–285, 1998.

[5] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In *Proc. IEEE 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, 2002.

[6] A. Briney. Information security 2000. *Information Security*, pages 40–68, 2000.

[7] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum, 1978.

[8] C. Date. *An Introduction to Database Systems*. Addison-Wesley, 2003.

[9] D. Ferraiolo, J. Cugini, and R. Kuhn. Role-based access control (RBAC): Features and motivations. In *Proc. of the 11th Annual Computer Security Applications Conf.*, pages 241–248, 1995.

[10] Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[11] B. Grosof and T. Poon. Representing agent contracts

with exceptions using xml rules, ontologies and process descriptions. In *WWW 2003*, pages 340–349, 2003.

[12] S. Jajodia, P. Samarati, M. Sapino, and V. Subrahmaninan. Flexible support for multiple access control policies. *ACM TODS*, 26(2):214–260, 2001.

[13] N. D. Jones. Partial evaluation, self-application and types. In M. S. Paterson, editor, *Automata, Languages and Programming*, LNCS 443, pages 639–659. Springer-Verlag, 1990.

[14] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[15] M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specializing interpreters using offline partial deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS 3049, pages 341–376. Springer-Verlag, 2004.

[16] M. Leuschel, J. Jorgensen, W. VanHoof, and M. Bruynooghy. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.

[17] M. Leuschel and D. D. Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *JLP*, 36(1):149–193, 1998.

[18] H. Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, LNCS 1924, pages 129–148. Springer-Verlag, 2000.

[19] K. Marriott and P. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[20] T. Przymusinski. On the declarative semantics of deductive databases and logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan-Kaufmann, 1988.

[21] K. Sagonas, T. Swift, D. Warren, J. Freire, and P. Rao. *The XSB System Version 2.0, Programmer's Manual*, 1999.

[22] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.

[23] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proc. 4th ACM Workshop on Role-Based Access Control*, pages 47–61, 2000.

APPENDIX

A. SPECIALISED INTERPRETER FOR Q_3

A series of tests are undertaken in Section 5, showing the efficiency of $RBAC^P_{H2A}$ programs after specialising our interpreter using LOGEN. In this section we provide the actual code obtained from the specialiser, for both aggressive and non-aggressive specialisation. Annotating the *senior_to* clause of `permitted/3` as a *rescall*, i.e., adding it to the residual code so that it is not evaluated in the specialisation process, leads to the non-aggressive approach shown below:

```
bench__0 :-
        ensure_loaded(database_cycle),
        abolish_all_tables, cputime(A),
        b2__1,
        cputime(B), C is B-A, print(C), nl.
b2__1 :-
        seniorto(r1, r53),
        holds_read_rule__2(_, _),
        fail.
b2__1.
:- table holds_read_rule__2/2.
holds_read_rule__2(A, B) :-
        seniorto(r1, r53),
        p(A, B).
holds_read_rule__2(A, B) :-
        seniorto(r1, r53),
        holds_read_rule__2(A, C),
        seniorto(r1, r53),
        p(C, B).
```

By considering the *senior_to* clause as *unfold*, i.e., being computed in the specialisation, the resulted program may be more efficient, at the expense of having to re-specialise each time a parameter to this clause changes. This slightly more aggressive result is shown as follows:

```
bench__0 :-
        ensure_loaded(database_cycle),
        abolish_all_tables, cputime(A),
        b2__1,
        cputime(B),
        C is B-A, print(C), nl.
b2__1 :-
        holds_read_rule__2(_, _),
        fail.
b2__1.
:- table holds_read_rule__2/2.
holds_read_rule__2(A, B) :-
        p(A, B).
holds_read_rule__2(A, B) :-
        holds_read_rule__2(A, C),
        p(C, B).
```

Observe that `holds_read_rule__2` is isomorphic to the `cycle` predicate, hence Jones-optimality [13, 14, 18, 15] has been achieved.

B. THE FULL INTERPRETER

Below is the full code of the interpreter, including a predicate `bench` used for benchmarking our query $Q3$. The predicates for queries $Q1$, $Q2$, and $Q4$ are very similar. The code below is intended for XSB Prolog, minor modifications were done for SICStus (e.g., replacing `cputime` by `statistics`).

```
ura(steve,r1).

active(steve,r1).

pra(r53,read,p(_,_)).
pra(r53,read,cycle(_,_)).
pra(r53,read,tcp(_,_)).
pra(r53,read,q(_)).

:- table holds_read/2.

holds_read(User,not(Object)) :-
  \+(holds_read(User,Object)).
holds_read(_User,Object) :- built_in(Object).
holds_read(User,Object) :-
  permitted(User,read,Object), fact(Object),
  call(Object).
```

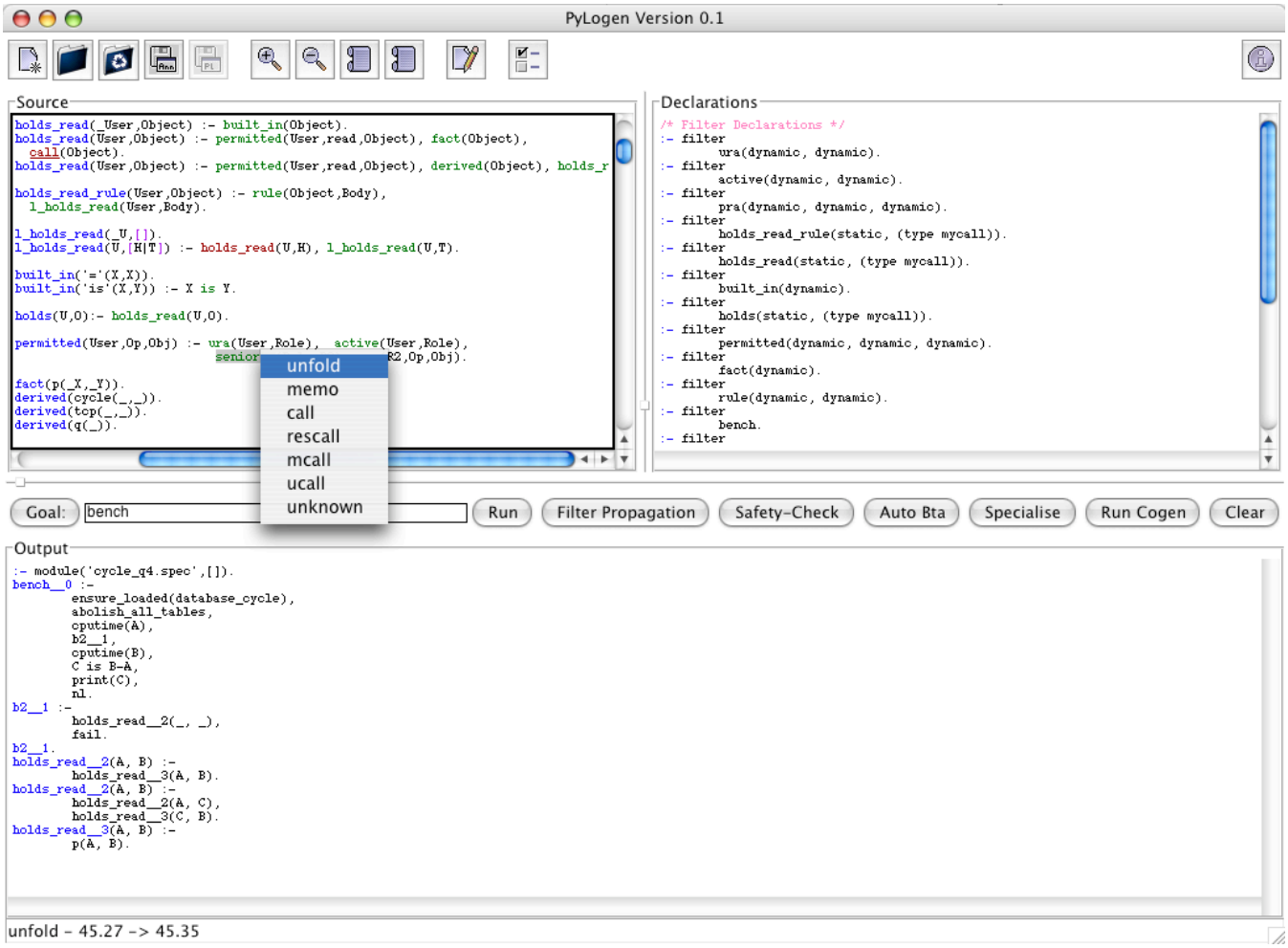

Figure 2: Snapshot of a LOGEN session

```
holds_read(User,Object) :-
  permitted(User,read,Object), derived(Object),
  holds_read_rule(User,Object).

holds_read_rule(User,Object) :- rule(Object,Body),
  l_holds_read(User,Body).

l_holds_read(_U,[]).
l_holds_read(U,[H|T]) :-
  holds_read(U,H),
  l_holds_read(U,T).

built_in('='(X,X)).
built_in('is'(X,Y)) :- X is Y.

holds(U,O):- holds_read(U,O).

permitted(User,Op,Obj) :-
  ura(User,Role),  active(User,Role),
  seniorto(Role,R2), pra(R2,Op,Obj).

fact(p(_X,_Y)).
derived(cycle(_,_)).
derived(tcp(_,_)).
derived(q(_)).

rule(cycle(X1,X2),[p(X1,X2)]).
rule(cycle(X1,X2),[cycle(X1,X3),p(X3,X2)]).
rule(tcp(X1,X2),[p(X1,X2)]).
rule(tcp(X1,X2),[p(X1,X3),tcp(X3,X2)]).
rule(q(X),[p(X,Y),not(q(Y))]).

% For benchmarking query Q3:
b2 :- holds_read(steve,cycle(_,_)),fail.
b2.
bench :- ensure_loaded('database_cycle'),
    abolish_all_tables, cputime(T1),
    b2,
    cputime(T2), R is T2-T1, print(R),nl.
```

C. THE ANNOTATED INTERPRETER

Bellow is the annotation file as used in the experiments (aggressive settings). Our GUI interface (see Appendix D) enables users to view and edit this file in a friendly fashion.

Note that the use of the `nonvar` annotation was essential to obtain good specialisation results (see also [15]). Also observe that a custom type `mycall` was added so as to avoid throwing away information within a negated call.

```
logen(ura, ura(steve,r1)).
logen(active, active(steve,r1)).
logen(pra, pra(r53,read,p(_,_))).
logen(pra, pra(r53,read,cycle(_,_))).
```

```
logen(pra, pra(r53,read,tcp(_,_))).
logen(pra, pra(r53,read,q(_))).
(:-true).
logen(holds_read, holds_read(A,not(B))) :-
        resnot(logen(memo,holds_read(A,B))).
logen(holds_read, holds_read(_,A)) :-
        logen(unfold, built_in(A)).
logen(holds_read, holds_read(A,B)) :-
        logen(unfold, permitted(A,read,B)),
        logen(unfold, fact(B)),
        logen(rescall, call(B)).
logen(holds_read, holds_read(A,B)) :-
        logen(unfold, permitted(A,read,B)),
        logen(unfold, derived(B)),
        logen(unfold, holds_read_rule(A,B)).
logen(holds_read_rule, holds_read_rule(A,B)) :-
        logen(unfold, rule(B,C)),
        logen(unfold, l_holds_read(A,C)).
logen(l_holds_read, l_holds_read(_,[])).
logen(l_holds_read, l_holds_read(A,[B|C])) :-
        logen(memo, holds_read(A,B)),
        logen(unfold, l_holds_read(A,C)).
logen(built_in, built_in(A=A)).
logen(built_in, built_in(A is B)) :-
        logen(unfold, A is B).
logen(holds, holds(A,B)) :-
        logen(unfold, holds_read(A,B)).
logen(permitted, permitted(A,B,C)) :-
        logen(unfold, ura(A,D)),
        logen(unfold, active(A,D)),
        logen(unfold, seniorto(D,E)),
        logen(unfold, pra(E,B,C)).
logen(fact, fact(p(_,_))).
logen(derived, derived(cycle(_,_))).
logen(derived, derived(tcp(_,_))).
logen(derived, derived(q(_))).
logen(rule, rule(cycle(A,B),[p(A,B)])).
logen(rule, rule(cycle(A,B),[cycle(A,C),p(C,B)])).
logen(rule, rule(tcp(A,B),[p(A,B)])).
logen(rule, rule(tcp(A,B),[p(A,C),tcp(C,B)])).
logen(rule, rule(q(A),[p(A,B),not(q(B))])).

logen(b2, b2) :-
        logen(memo, holds_read(steve,cycle(_,_))),
        logen(rescall, fail).

logen(b2, b2).
logen(bench, bench) :-
        logen(rescall, ensure_loaded(database_cycle)),
        logen(rescall, abolish_all_tables),
        logen(rescall, cputime(A)),
        logen(unfold, b2),
        logen(rescall, cputime(B)),
        logen(rescall, C is B-A),
        logen(rescall, print(C)),
        logen(rescall, nl).

:- type mycall = (not(nonvar) ; nonvar).
:- filter ura(dynamic, dynamic).
:- filter active(dynamic, dynamic).
:- filter pra(dynamic, dynamic, dynamic).
:- filter holds_read_rule(static, type(mycall)).
:- filter holds_read(static, type(mycall)).
:- filter built_in(dynamic).
:- filter holds(static, type(mycall)).
:- filter permitted(dynamic, dynamic, dynamic).
:- filter fact(dynamic).
:- filter rule(dynamic, dynamic).
```

D. THE LOGEN TOOL IN ACTION

The purpose of this section is more of an illustrative nature rather than actually presenting any theoretical nor empirical result. In this section we show the actual phase of specialising the $RBAC^P_{H2A}$ program by means of the LOGEN system. As shown in Figure 2, LOGEN is built with a Graphical User Interface (GUI) which facilitates the specialisation of logic programs. This snapshot illustrates how $RBAC^P_{H2A}$ program were annotated and specialised in the context of this framework. On the one hand, the *source code* of the database meta-interpreter (left window) is annotated by unrolling a list of options for each clause in a predicate. On the other, *filter declarations* are typed in the far right window, in order to guide the specialisation process. This allows LOGEN to, after just pressing a button, specialise the program for which the $RBAC^P_{H2A}$ policy has been optimised towards a particular query (i.e., the program is specialised according to the IDB of Δ).