Journal of Technical Science & Technologies; ISSN: 2298-0032; e-ISSN: 2346-8270; Volume 5, Issue 2, 2016

PρLog for Access Control

Besik DUNDUA*
Khimuri RUKHAIA**
Mikheil RUKHAIA***
Lali TIBUA****

Abstract

In this article we show how access control policies can be expressed in PρLog, which is a system for programming with conditional transformation rules, controlled by strategies. PρLog combines the power of logic programming with rewriting, which makes it convenient to reason about the policies.

Keywords: Access control, programming with strategies, PρLog

Introduction

In recent years usefulness of sequence and context variables has been shown in various areas of mathematics and computer science. Sequence variables are placeholders for arbitrarily long finite sequences of expressions and have applications in programming (Wolfram, 2003), XML querying and transformation (Coelho, Dundua, Florido, & Kutsia, 2010), knowledge engineering and artificial intelligence (Volpano, 1994), and automated reasoning (Ginsberg, 1991). Context variables are placeholders for contexts, which are functional expressions whose applicative behavior is to replace a special constant (so called hole) with the expression given as argument. They have applications in compositional semantics of natural language (Koller, 1998). Combination of these variables together with individual and function variables in a single framework allows flexible term traversal in arbitrary width (with individual and sequence variables) and in arbitrary depth (with function and context variables).

PρLog is a system (Dundua, 2014; Dundua, Kutsia, & Marin, 2009) that supports programming with individual, sequence, function, and context variables. It extends Prolog with rule based programming capabilities to manipulate the sequences of trees, also known as sequences. PρLog has a computational model based on ρLog calculus (Marin & Kutsia, 2006) and a logic programming semantics where program clauses define strategies that act on sequences. Such an extension allows to bring the whole Prolog power into PρLog and adds rule based strategic programming capabilities to it. Strategies work on sequences and provide a mechanism to control complex rule based computations in a highly declarative way. Such an extension of Prolog is expressive enough to support concise implementations for specifying and prototyping deductive systems, solvers for various equational theories, etc. PρLog system has been used in XML transformation and Web reasoning (Coelho, et al., 2010), in evaluation strategies (Dundua, et al., 2009), and in extraction of frequent patterns from data mining workflows (Nguyen, 2015). In this paper, we show how access control policies can be expressed in PρLog.

Access control is a security technique that specifies which users can access particular resources in a computing environment. Over the years, numerous access control models have been developed to address various aspects of computer security. In this work, we focus on the role-based access control (RBAC) (Ferraiolo, Sandhu, Gavrila, Kuhn, & Chandramouli, 2001), which has been proposed in order to overcome limitations of traditional models: discretionary access control (DAC) (Sandhu & Samarati, 1994) and mandatory access control (MAC) (Sandhu, 1993). Despite successful practical applications of these traditional models, they have certain disadvantages, which was the reason why new approaches emerged.

* Ph.D., Institute of Applied Mathematics, Tbilisi State University, & Faculty of Computer Technologies and Engineering, International Black Sea University, Tbilisi, Georgia.
E-mail: bdundua@gmail.com
** Assoc. Prof. Dr., Faculty of Exact and Natural Sciences, Tbilisi State University,& Faculty of Mathematics and Computer Sciences, Sokhumi State University, Tbilisi, Georgia.
E-mail: khimuri.rukhaia@gmail.com
*** Assoc. Prof. Dr., Institute of Applied Mathematics, Tbilisi State University, & Faculty of Computer Technologies and Engineering, International Black Sea University, Tbilisi, Georgia.
E-mail: mrukhaia@ibsu.edu.ge
**** Ph. D., Faculty of Exact and Natural Sciences, Tbilisi State University, & School of IT, Engineering and Mathematics, University of Georgia, Tbilisi, Georgia.
E-mail: ltibua@gmail.com

The main problem with DAC is that it lacks control over the flow of information (information is copied from one object to another). MAC solves this problem by classifying subjects and objects with respect to their security levels, granting request by a subject to an object if a relationship between the subjects security level and the object is satisfied. However, once security levels are assigned, they cannot be changed. RBAC is designed to be a more general model than MAC or DAC where users are assigned a certain set of roles. The roles, in turn, have permissions assigned to them. A request made by a user is authorized, if the roles assigned to the user contain permissions that allow the request. RBAC has found its applications in many commercial and governmental sectors (Gilbert, 1995).

In this paper we demonstrate application of the PρLog system in access control. In particular, using an example we illustrate how role based access control problem can be encoded in the PρLog system as well as how the PρLog can be used for evaluating access requests.

The PρLog Language

PρLog's strategic conditional transformation rules are supposed to transform term sequences when the condition is satisfied. Strategies can be combined by means of a language of strategy operators to express many tedious small step transformations in a compact way and provide mechanism to control complex rule-based computation in a highly declarative way.

A strategic conditional transformation rule, which swaps adjacent elements in a sequence if they satisfy a given condition can be implemented in PρLog as follows:

```
swap::   (s_X,i_x,i_y,s_Y) ==>
         (s_X,i_y,i_x,s_Y) :- i_x > i_y.
```

Here swap is the strategy name. It is followed by the separator :: which separates the strategy name from the transformation. Then comes the transformation itself in the form lhs ==> rhs. It says that if the sequence in lhs contains neighboring elements (i_x and i_y) such that i_x > i_y then swap them in rhs. Here :-, as in Prolog, stands for the inverse implication. Individual variables (i_x and i_y) are used to select terms in a sequence, while sequence variables s_X and s_Y match subsequences before i_x and after i_y respectively.

Now one can ask a question, e.g., to swap neighboring elements in a sequence of numbers (1,3,2,1):

```
?- swap::(1,3,2,1) ==>  s_Result.
```

The sequence (s_X, i_x, i_y, s_Y) matches (1,3,2,1) in three different ways, as indicated by the following mappings:

1. s_X → () (indicating that s_X matches the empty sequence), i_x → 1, i_y → 3, s_Y → (2,1) (meaning that s_Y matches the sequence 2,1) .

2. s_X → 1, i_x → 3, i_y → 2 and s_Y → 1.

3. s_X → (1,3), i_x → 2, i_y → 1 and s_Y → ().

In the first case, the condition i_x > i_y is not satisfied and query evaluation fails, while the second and third cases satisfy the condition i_x > i_y and, via backtracking, PρLog returns two answers s_Result → (1,2,3,1) and s_Result → (1,3,1,2).

swap was an instance of a user-defined strategy. PρLog provides built-in strategies as well. Both user-defined and built-in strategies can be used in programs and queries, they can participate in constructing more complex strategies, etc. The difference is that built-in strategies are fixed by the system and the user cannot redefine them. We give a brief overview some of the built-in strategies which will be used later in expressing access control policies.

Identity. The goal of this strategy is to transform a sequence to its identical one:

```
id :: sequence1 ==> sequence2.
```

It succeeds iff sequence2 can match sequence1.

Composition. Composing strategies, so that the output sequence of one becomes the input for the other:

```
compose(strategy1,...,strategyn)::
        sequence1 ==> sequence2,
```

where n ≥ 2. First applies strategy1 to sequence1. To its result, strategy2 is applied and so on. sequence2 is the final result. compose fails if one of its argument strategies fails in the process.

Returning first answer of the first applicable strategy. Denoted by first_one:

```
first_one(strategy1,...,strategyn)::
        sequence1 ==> sequence2,
```

where n ≥ 1. Tries to apply strategy1 to sequence1. If this fails, it tries the next strategy and so on. When a strategy is found that succeeds, first_one returns first answer computed by it in sequence2. If no strategy succeeds, first_one fails.

Normal form. Applies repeatedly a strategy to a given sequence, before normal form is reached:

```
nf(strategy)::sequence1 ==> sequence2.
```

It applies strategy repeatedly to sequence1 until the transformation is not possible, and returns the last sequence. The result is returned in sequence2.

Mapping a strategy to a sequence. It is a common operation in declarative programming. The corresponding PρLog strategy is

```
map(strategy)::sequence1 ==> sequence2.
```

It applies strategy to each term of sequence1. For such an input term, strategy may, in general, return a sequence (not necessarily a single term). A sequence constructed of these results (in the same order) is then returned in sequence2. map fails when the application of strategy to a term from sequence1 fails. When sequence1 is empty, sequence2 is empty as well.

The syntax of PρLog is largely similar to Prolog. The PρLog variables are Prolog constants. Besides already mentioned individual and sequence variables, there are also function and context variables in PρLog that help to perform a certain type of second-order matching. In general, variable names start with the first letter of their kind, followed by the underscore. After the underscore, there comes the actual name. We have already seen the examples of two kinds of variables: individual variables i_x and i_y, and sequence variables s_X,s_Y,s_Result. The syntax of atomic strategies has also been explained at the beginning of this section. For a reference, a more detailed summary of PρLog syntax is given in Fig. 1.

Role Based Access Control

The simplest model of RBAC consists of sets of users (U), roles (R), and permissions (P). An user $u \in U$ in this model is a human being assigned to a set of roles $\{r_1,...,r_n\} \subseteq R$. Permissions is an approval of a certain mode of access to an object. The following example illustrates modeling of a simple RBAC scenario in PρLog.

Example 3.1 Suppose that the user u1 is assigned to the roles r1 and r2, and the user u2 is assigned to the role r2. Moreover, suppose that the role r1 is assigned the reading permission on the object o1 and the writing permission on the object o2, and the role r2 is assigned the writing permission on o1. The user-role and role-permission assignments in PρLog are expressed as follows:

```
              clause ::= ρ-clause | strategy_definition |
                         Prolog clause
          meta-query ::= ?(query, Prolog variable).
               query ::= literal | literal, query
            ρ-clause ::= ρ-atom. | ρ-atom :- query.
              ρ-atom ::= strategy :: (sequence) ==>
                         (sequence) [where constraint]
             literal ::= ρ-literal | Prolog literal
           ρ-literal ::= ρ-atom |
                         strategy :: (sequence) =\=>
                         (sequence) [where constraint]
            strategy ::= ρ-term
 strategy_definition ::= strategy := strategy
            sequence ::= eps | nonempty_sequence
   nonempty_sequence ::= term_or_seq_var |
                         term_or_seq_var, nonempty_sequence
     term_or_seq_var ::= ρ-term | seq_var
              ρ-term ::= ind_var | fun_var(sequence) |
                         Prolog constant(sequence) |
                         ctx_var(ρ-term)
             context ::= hole | fun_var(sequence, context, sequence) |
                         Prolog constant(sequence, context, sequence) |
                         ctx_var(context)
             ind_var ::= Prolog constants that start with i_
             seq_var ::= Prolog constants that start with s_
             fun_var ::= Prolog constants that start with f_
             ctx_var ::= Prolog constants that start with c_
        substitution ::= List of bindings
             binding ::= ind_var ---> ρ-term |
                         seq_var ---> (sequence) |
                         fun_var ---> Prolog constant |
                         ctx_var ---> context
```

Figure 1. *A summary of PρLog syntax. The symbols* ***eps*** *and* ***hole*** *are the notation for the empty sequence and the hole, respectively.*

```
user_role :: u1 ==> (r1,r2).
user_role :: u2 ==> r2.
role_permissions :: r1 ==>
    (read(o1),write(o2)).
role_permissions :: r2 ==> write(o1).
```

If a role is assigned to have a write permission to an object, it is obvious that the role has read permission to that object as well. We encode, this property in PρLog as follows:

```
implies :: write(i_Object) ==>
    (read(i_Object),write(i_Object)) :−
    !.
implies :: i_Privilege ==> i_Privilege.
```

Each role can be assigned several permissions and each user can have several roles, which can be encoded in PρLog as follows:

```
access :: (i_User,i_Obj) ==> grant :−
    all_permissions_for_a_user ::
    i_User ==> (s__,i_Obj,s__),
    !.
access :: (i_User,i_Obj) ==> deny.
all_permissions_for_a_role :=
    compose(role_permissions,
    map(implies)).
all_permissions_for_a_user :=
    compose(
    all_answers_flat(compose(user_role,
    map(all_permissions_for_a_role))),
    merge_all_doubles).
merge_doubles :: (s_X,i_X,s_Y,i_X,s_Z)
    ==> (s_X,i_X,s_Y,s_Z).
merge_all_doubles :=
    first_one(nf(merge_doubles)).
all_answers_flat(i_Strategy) :=
    compose(all_answers(i_Strategy),
    map(flatten_ans)).
flatten_ans :: ans(s_X) ==> s_X.
```

Now we can start asking questions to this program. For instance, to find out whether u1 has the read permission on o1:

```
?(access::(u1,read(o1)) ==> i_X, Subst).
    Subst = [i_X--->grant]
```

The answer substitution says that u1 should be granted the read permission on o1. Answers to the queries below are interpreted similarly.

```
?(access::(u1,write(o1))==> i_X, Subst).
        Subst = [i_X--->grant]
?(access::(u1,read(o2)) ==> i_X, Subst).
        Subst = [i_X--->grant]
?(access::(u1,write(o2))==> i_X, Subst).
        Subst = [i_X--->grant]
?(access::(u2,read(o1)) ==> i_X, Subst).
        Subst = [i_X--->grant]
?(access::(u2,write(o1))==> i_X, Subst).
        Subst = [i_X--->grant]
?(access::(u2,read(o2)) ==> i_X, Subst).
        Subst = [i_X--->deny]
?(access::(u2,write(o2))==> i_X, Subst).
        Subst = [i_X--->deny]
```

One can easily check that the retuned answers are all expected. In general, access control policies should be consistent, i.e. the same user should not be at the same time granted and denied the same privilege to the same object. By analyzing the program, we can easily observe that first, there cannot be infinite computations, since there are no recursions and built-in strategies are terminating (the latter can be proved once and for all). Moreover, two clauses defining the access strategy are mutually exclusive: if a privilege is granted, then it cannot be denied and vice versa. These properties imply that on every request we get an answer (which means our access control policy in total) and we cannot get at the same time grant and deny (which means the policy is consistent).

Conclusion

We briefly described PρLog and gave an example that illustrates how access control policies can be implemented in this language. The style of programming by conditional transformation rules, controlled by strategies, makes it convenient both to express the policies and to reason about them.

Acknowledgments

This research has been supported by the Shota Rustaveli National Science Foundation under the grants FR/508/4-120/14 and YS15_2.1.2_70.

References

Coelho, J., Dundua, B., Florido, M., & Kutsia, T. (2010). A rule-based approach to XML processing and web reasoning. In P. Hitzler & T. Lukasiewicz (Eds.), RR (Vol. 6333, p. 164-172). Springer.

Dundua, B. (2014). Programming with sequence and context variables:foundations and applications (Unpublished doctoral dissertation). Universidade do Porto.

Dundua, B., Kutsia, T., & Marin, M. (2009). Strategies in PρLog. In M. Fernandez (Ed.), Proceedings ninth international workshop on reduction strategies in rewriting and programming, WRS 2009, Brasilia, Brazil, 28th june 2009. EPTCS, vol. 15, pp. 32–43.

Ferraiolo, D. F., Sandhu, R. S., Gavrila, S. I., Kuhn, D. R., & Chandramouli, R. (2001). Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur., 4(3), 224–274.

Gilbert, M. D. M. (1995). An examination of federal and commercial access control policy needs. In National computer security conference, 1993 (16th) proceedings: Information systems security: User choices (p. 107).

Ginsberg, M. L. (1991). The MVL theorem proving system. SIGART Bulletin, 2(3), 57-60.

Koller, A. (1998). Evaluating context unification for semantic underspecification. In Proceedings of the third esslli student session (pp. 188–199).

Marin, M., & Kutsia, T. (2006). Foundations of the rule-based system ρLog. Journal of Applied Non-Classical Logics, 16(1-2), 151-168.

Nguyen, P. (2015). Meta-mining: a meta-learning framework to support the recommendation, planning and optimization of data mining workflows (Unpublished doctoral dissertation). Department of Computer Science, University of Geneva.

Sandhu, R. S. (1993). Lattice-based access control models. IEEE Computer, 26(11), 9–19.

Sandhu, R. S., & Samarati, P. (1994). Access control: principle and practice. IEEE communications magazine, 32(9), 40–48.

Volpano, D. M. (1994). Haskell-style overloading is np-hard. In H. E. Bal (Ed.), Iccl (p. 88-94). IEEE Computer Society.

Wolfram, S. (2003). The Mathematica Book (5. ed.). Wolfram-Media.