Protune proof and filtering process

J. L. De Coi

October 23, 2006

Abstract

This document tries to investigate the theoretical background of the *Protune* proof and filtering process.

1 Actions

The *Protune* language allows a policy author to specify actions within the policy.

- Actions are dealt in a procedural fashion, i.e. there is a clear distinction between input and output parameters
- In order to trigger an action, its input parameters must be instantiated (i.e. they must be ground)
- The successful execution of an action results in asserting a new fact (i.e. in modifying the actual knowledge base by adding a new fact) stating that the action was executed successfully
- The (successful) execution of an action may trigger the execution of other actions (e.g. in case the output value of an action unifies with the last still not ground input parameter of another action)

`tuProlog` [1] provides a straightforward mechanism for executing actions within (an extended version of) Prolog. Tab. 1 compares (tu)Prolog and Java codes required to show an alert window.

If we want to avoid re-executing *myAction* in the same or in following proofs, we need to track its execution in the knowledge base by adding a new fact stating that *myAction* was executed successfully. The suitable code is provided in Tab. 2.

Notice that the `tuProlog` engine requires that the second rule follows the first one.

2 Proof process

Many PROLOG systems use SLD-resolution as refutation procedure [2], i.e. in order to prove a goal G

myAction :- java_object('javax.swing.JOptionPane', [], JOP), JOP ← showMessageDialog(_, 'Hello, world!').

import javax.swing.JOptionPane;
 	
void myAction(){
JOptionPane JOP = new JOptionPane();
JOP.showMessageDialog(null, "Hello, world!");
}

Table 1: (tu)Prolog and Java codes required to show an alert window.

- a sequence $G_0, ...G_{last}$ of goals
- a sequence $C_1, ...C_{last}$ of variants of program clauses
- a sequence $\theta_1, ...\theta_{last}$ of mgus

are looked for, such that $G_0 = G$, $G_{last} = \circ$ and $\forall G_i = \leftarrow A_1, ...A_j, ...A_m$

- θ_{i+1} is an mgu of A_j and the head of C_{i+1}
- $G_{i+1} = \leftarrow A_1, ...A_{j-1}, B_1, ...B_n, A_{j+1}...A_m$ where $B_1, ...B_n$ is the body of C_{i+1}

A well known result ensures that if a goal can be proven at all, the same goal can be proven by exploiting whatever Computation Rule, i.e. by exploiting whatever rule for selecting the literal A_j. On the contrary in the *Protune* system, because of the previously mentioned side-effects, the computation rule matters, as shown in the following example.

```
allow(access(book)) :- provisional(X), X=book.
provisional(X).evaluation:immediate :- ground(X).
```

If $provisional(X)$ is evaluated before $X = book$ the proof will fail, since X is not ground, otherwise the proof will succeed, since X is (becomes) ground.

This suggests that, in evaluating a clause's body, the proof attempt should not backtrack after the first literal's failure but go on with the other ones. As

myAction :- java_object('javax.swing.JOptionPane', [], JOP), JOP ← showMessageDialog(_, 'Hello, world!'), assert(performed(myAction)). myAction :- performed(myAction).

Table 2: (tu)Prolog code avoiding multiple executions of *myAction*.

soon as a literal succeeds a new attempt with the previously failed literals should be carried out until either each literal succeeds or each (not yet succeeded) literal fails. Tab. 3 sketches the algorithm which should be used in evaluating a clause's body.

```
evaluateBody([]).

evaluateBody([H—T]) :-
evaluateLiteral(H), !, evaluateBody(T).

evaluateBody([H—T]) :-
evaluateBody(T), evaluateLiteral(H).
```

Table 3: Algorithm evaluating a clause's body.

Moreover standard Prolog systems use the order of clauses in a program as the fixed order in which they are to be tried. This behaviour has big drawbacks also in the *Protune* system, as shown in the following example.

```
[r1] allow(access(book)) :- provisional(X, Y).
[r2] allow(access(book)) :- provisional(book, X), not true.
provisional(X, _).evaluation:immediate :- ground(X).
```

If $[r1]$ is evaluated before $[r2]$ the proof will fail, since X is not ground and after backtracking $[r2]$'s body contains the literal *not true*. Otherwise the proof will succeed, since

- the evaluation of $provisional(book, X)$ will assert a fact like $performed(provisional(book, someResult))$
- the evaluation of $[r2]$ will fail but
- after backtracking, because of the assertion of $performed(provisional(book, someResult))$, $[r1]$ will succeed

This suggests that, in proving a goal, the proof should start again from the beginning each time an action is (successfully) executed.

3 Execution trace

In this section we are going to show how the *Protune* engine works. We will present three execution traces, the first one based on the examples provided above and the other ones based on a real world example.

The *Protune* engine works like a standard Prolog engine, but uses the two rules stated above.

Rule 1 *In evaluating a clause's body the algorithm described in Tab. 3 will be exploited.*

Rule 2 *The proof of a goal will start again from the beginning each time an action is (successfully) executed.*

3.1 First example

In this example we will consider the following policy.

```
allow(access(book)) :-
   provisional(X, Y).

allow(access(book)) :-
   provisional(X, Y),
   X=book,
   not true.

provisional(X, _).evaluation:immediate :- ground(X).
```

All steps of the proof of goal $?-\ allow(access(book))$. are provided in the following.

- A proof for goal $?-\ allow(access(book))$. is looked for
  - According to SLD-resolution the goal becomes $?-\ provisional(X, Y)$.
    - X is not ground, therefore the goal fails and backtracking is performed
  - The goal becomes $?-\ provisional(X, Y), X = book, not\ true$.
    - $provisional(X, Y)$ fails. According to Rule 1 a proof for $X = book$ is looked for
    - $X = book$ succeeds and the goal becomes $?-\ provisional(book, Y)$, $not\ true$.
    - $provisional(book, Y)$ succeeds and $performed(provisional(book, someResult))$ is asserted

An action was executed (resulting in asserting $performed(provisional(book, someResult))$), therefore according to Rule 2 the proof starts again from the beginning.

- A proof for goal $?-\ allow(access(book))$. is looked for
  - The goal becomes $?-\ provisional(X, Y)$.
    - $provisional(X, Y)$ succeeds since it matches with the asserted fact $performed(provisional(book, someResult))$

3.2 Second example

This example is adapted from [3]. We will consider the following policy.

```
allow(access(Resource)) :-
   credential(sa, Student_card[type: student, issuer:I,
      public_key: K]),
   valid_credential(Student_card, I),
   is_recognized_university(I),
   challenge(K).

allow(access(Resource)) :-
   authenticate(U),
   has_subscription_for(U, Resource).

valid_credential(C, I) :-
   get_public_key(I, K),
   verify_signature(C, K).

authenticate(U) :-
   declaration(ad, D[username: U, password: P]),
   passwd(U, P).

is_recognized_university(X).evaluation:immediate :- ground(X).
challenge(X).evaluation:immediate :- ground(X).
has_subscription_for(X, Y).evaluation:immediate :- ground(X),
   ground(Y).
get_public_key(X, _).evaluation:immediate :- ground(X).
verify_signature(X, Y).evaluation:immediate :- ground(X),
   ground(Y).
passwd(X, Y).evaluation:immediate :- ground(X), ground(Y).
```

All steps of the proof of goal ? − $allow(access(book))$. are provided in the following.

- A proof for goal ? − $allow(access(book))$. is looked for
  - According to SLD-resolution the goal becomes ? − $credential(...)$, $valid\_credential(Student\_card, I)$, $is\_recognized\_university(I)$, $challenge(K)$.
    - The Client did not provided any credential yet, therefore $credential(...)$ fails. According to Rule 1 a proof for $valid\_credential(Student\_card, I)$ is looked for
    - According to SLD-resolution $valid\_credential(Student\_card, I)$ becomes $get\_public\_key(I, K), verify\_signature(Student\_card, K)$
      - I is not ground, therefore $get\_public\_key(I, K)$ fails. According to Rule 1 a proof for $verify\_signature(Student\_card, K)$ is looked for

        - Neither $Student\_card$ nor K are ground, therefore $verify\_signature(Student\_card, K)$ fails
    - As just shown, $valid\_credential(Student\_card, I)$ fails. According to Rule 1 a proof for $is\_recognized\_university(I)$ is looked for
    - I is not ground, therefore $is\_recognized\_university(I)$ fails. According to Rule 1 a proof for $challenge(K)$ is looked for
    - $challenge(K)$ is not ground, therefore the goal fails and backtracking is performed
- According to SLD-resolution the goal becomes $?-$ $authenticate(U)$, $has\_subscription\_for(U, Resource)$.
    - According to SLD-resolution $authenticate(U)$ becomes $declaration(...)$, $passwd(U, P)$
        - The Client did not provided any declaration yet, therefore $declaration(...)$ fails. According to Rule 1 a proof for $passwd(U, P)$ is looked for
        - Neither U nor P are ground, therefore $passwd(U, P)$ fails
    - As just shown, $authenticate(U)$ fails. According to Rule 1 a proof for $has\_subscription\_for(U, Resource)$ is looked for
    - Neither U nor $Resource$ are ground, therefore the goal fails

3.3 Third example

We will consider the same policy of the previous example, but this time we will suppose that the Client has already sent the credential

$$credential(sa, studentcard[type : student, issuer : hu, public\_key : 5272117]).$$

All steps of the proof of goal $?-$ $allow(access(book))$. are provided in the following.

- A proof for goal $?-$ $allow(access(book))$. is looked for
    - According to SLD-resolution the goal becomes $?-$ $credential(...)$, $valid\_credential(Student\_card, I)$, $is\_recognized\_university(I)$, $challenge(K)$.
        - $credential(...)$ succeeds, since the Client provided a credential. The goal becomes $?-$ $valid\_credential(studentcard, hu)$, $is\_recognized\_university(hu)$, $challenge(5272117)$.
        - According to SLD-resolution $valid\_credential(studentcard, hu)$ becomes $get\_public\_key(hu, K)$, $verify\_signature(studentcard, K)$
            - $get\_public\_key(hu, K)$ succeeds and $performed(get\_public\_key(hu, someResult))$ is asserted.

An action was executed (resulting in asserting $performed(get\_public\_key(hu, someResult))$), therefore according to Rule 2 the proof starts again from the beginning.

- Cf. above
  - Cf. above
    - Cf. above
      - $get\_public\_key(hu, K)$ succeeds since it matches with the asserted $performed(get\_public\_key(hu, someResult))$
      - $verify\_signature(studentcard, someResult)$ succeeds and $performed(verify\_signature(studentcard, someResult))$ is asserted.

An action was executed (resulting in asserting $performed(verify\_signature(studentcard, someResult))$), therefore according to Rule 2 the proof starts again from the beginning.

- Cf. above
  - Cf. above
    - Cf. above
      - Cf. above
      - $verify\_signature(studentcard, someResult)$ succeeds since it matches with the asserted $performed(verify\_signature(studentcard, someResult))$)
    - As just shown, $valid\_credential(studentcard, hu)$ succeeds. The goal becomes $?- is\_recognized\_university(hu), challenge(5272117)$.
    - $is\_recognized\_university(hu)$ succeeds and $performed(is\_recognized\_university(hu))$ is asserted.

An action was executed (resulting in asserting $performed(is\_recognized\_university(hu))$), therefore according to Rule 2 the proof starts again from the beginning.

- Cf. above
  - Cf. above
    - Cf. above
    - $is\_recognized\_university(hu)$ succeeds since it matches with the asserted $performed(is\_recognized\_university(hu))$. The goal becomes $?- challenge(5272117)$.
    - $challenge(5272117)$ succeeds and $performed(challenge(5272117))$ is asserted.

An action was executed (resulting in asserting $performed(challenge(5272117))$), therefore according to Rule 2 the proof starts again from the beginning.

- Cf. above
  - Cf. above
    - Cf. above
    - *challenge*(5272117) succeeds since it matches with the asserted $performed(challenge(5272117))$. Therefore the overall goal succeeds.

4 Filtering process

Definition 1 (Semantics-preserving filtered policy [4]) *A semantics-preserving filtered policy is the set of (facts and) rules used during the (last) proof process.*

Dynamically asserted facts should be considered as well. Tab. 4 shows the semantics-preserving filtered policies related to the three previous examples.

The filtering process may also involve information loss. Tab. 5 shows all components a policy can consist of.

Notice that

- according to [4] only provisional predicates can be declared to be executed either by the current Peer or by the other one
- it is pointless adding a provisional predicate to be executed by the other Peer in the definition of a private abbreviation predicate, since the other Peer will never be aware of such definition
- only abbreviation predicates need to be declared either *public* other *private*, indeed
  - it is pointless declaring a provisional predicate to be executed by the other Peer as *public* or *private* (it is the other Peer's business)
  - it is not needed declaring a provisional predicate to be executed by the current Peer as *public* or *private*, since the same result can be obtained by wrapping it with a suitable abbreviation predicate (i.e. by declaring the provisional predicate as the definition of a suitable abbreviation predicate). Notice that wrapping the same provisional predicate with abbreviation predicates having different sensibility will result in varying the sensibility of the provisional predicate itself according to the current context. An example will maybe help in understanding the background idea

Example 1 *If provisional() must be considered sometimes public and sometimes private, it suffices defining the two predicates*

```
publicAbbreviation(<parameters>) :- provisional(<parameters>).
privateAbbreviation(<parameters>) :- provisional(<parameters>).
```

and, instead of directly using provisional(), using from time to time publicAbbreviation() or privateAbbreviation() according to the context.

References

[1] D.E.I.S. (2005) *tuProlog Developers Guide*, Version 1.3.0

[2] J. W. Lloyd (1987) *Foundations of Logic Programming*, 2nd Edition

[3] D. Ghita (2006) *Policy Adaptation and Exchange in Trust Negotiation*

[4] P. A. Bonatti, D. Olmedilla (2005) *Policy Language Specification*

allow(access(book)) :- provisional(X, Y). performed(provisional(book, someResult)).
allow(access(Resource)) :- credential(sa, Student_card[type: student, issuer:I, public_key: K]), valid_credential(Student_card, I), is_recognized_university(I), challenge(K). allow(access(Resource)) :- authenticate(U), has_subscription_for(U, Resource). valid_credential(C, I) :- get_public_key(I, K), verify_signature(C, K). authenticate(U) :- declaration(ad, D[username: U, password: P]), passwd(U, P).
allow(access(Resource)) :- credential(sa, Student_card[type: student, issuer:I, public_key: K]), valid_credential(Student_card, I), is_recognized_university(I), challenge(K). valid_credential(C, I) :- get_public_key(I, K), verify_signature(C, K). credential(sa, studentcard[type: student, issuer: hu, public_key: 5272117]). performed(get_public_key(hu, someResult)). performed(verify_signature(studentcard, someResult)). performed(is_recognized_university(hu)). performed(challenge(5272117)).

Table 4: Semantics-preserving filtered policies examples.

```
allow(access(Resource)) :-
publicAbbreviation(¡parameters¿),
privateAbbreviation(¡parameters¿),
selfProvisional1(¡parameters¿),
otherProvisional(¡parameters¿).

publicAbbreviation(¡parameters¿) :-
selfProvisional2(¡parameters¿),
otherProvisional2(¡parameters¿).

privateAbbreviation(¡parameters¿) :-
selfProvisional3(¡parameters¿).
```

Table 5: Basic components of a policy.