Activity report

J. L. De Coi

November 10, 2006

Abstract

This document aims at describing how the Prolog part of the *Protune* system was realised.

1 Overall strategy

The filtering process described in [1] requires that the (Prolog) interpreter of the *Protune* language directly manages variable instantiation/unification (cf. [1], pg. 48). According to [2] such a fine-grained interpreter can be realised, but

- it is not easy (cf. [2] pgg. 326, 329)
- working at this fine level is usually not worthwhile, because of efficiency loss (cf. [2] pg. 326)

Since avoiding the efficiency loss would require building a new Prolog interpreter from the schratch (which is out of scope as well), maybe it could be worth redefining the filtering process such in a way that

- the result of the new filtering process is as close as possible to the one described in [1]
- the new filtering process is not so hard to implement and can be efficiently executed

In the following we are going to describe this new (and already implemented) filtering process.

2 Filtering process

The filtering process described in [1] combines in a single procedure three (conceptually) distinct operations

1. extraction of the actions the current peer has to execute in order to advance the negotiation

2. proof of the peer request

3. actual filtering of the current peer's policy to be sent to the other one

The filtering process we propose keeps the previous steps distinct. The steps are executed in the shown order. Moreover, since the (Java-side) action execution may trigger the execution of other actions, the first step will be repeated (and the collected actions executed) until no more actions can be extracted.

Note 1 *When we are going to connect the Prolog- and Java-parts of the Protune system, we will need to decide whether the filtered policy should be sent to the peer even if the negotiation was already successful.*

2.1 Extracting actions

During the first step the policy is scanned in order to find

- all provisional predicates
- with actor `self`
- which were not evaluated yet and
- which can be immediately evaluated (i.e. whose evaluation is at present `immediate`)

The scan starts with the (definition of the) decision predicates matching the peer request. If during the scan an abbreviation predicate is found, its definitions are scanned as well.

Note 2 *At present the algorithm extracting actions gets looped when recursive rules are encountered. As soon as possible I will correct this tedious bug.*

2.2 Proof of the peer request

The proof of the peer request is a normal Prolog proof. Notice that this proof cannot fail because some provisional predicates were not executed by the current peer, since the (possible iteration of the) previous step ensures that the current peer executed all actions it could.

2.3 Filtering process

During the filtering step the policy is scanned once again. The scan starts with the (definition of the) decision predicates matching the peer request. If during the scan a public abbreviation predicate or a public state query predicate is found, its definitions are scanned as well.

If appearing in the body of some rule

- private abbreviation predicates and private state query predicates are blurred
- provisional predicates with actor `self` and already executed provisional predicates with actor `peer` are filtered out

Note 3 *In [1] provisional predicates with actor `self` are not filtered out (if private, they are just blurred). I do not think that the other peer will be interested in the actions the current peer should carry out. But even if it were the case, changing this part of the code will require no more than one minute.*

3 Examples

In the following we are going to explain how the filtering process we proposed works by means of two different examples. The first one is the one provided in [1], it will turn out that the result of our filtering is (almost) the same than the one described in [1]. The second example was meant to be as general as possible and it should show how our filtering process works in every circumstance.

3.1 First example

The (meta)policy this example bases on can be found in [1], starting from page 84. The peer request is `allow(access(books))` and the peer already sent credential

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

Extracting actions (First iteration) The peer request matches `[r1]`, `[r2]` and `[r4]`. After unification with the credential sent by the peer, `[r1]`'s body contains the abbreviation predicate `valid_credential(studentcard, hu)` and the provisional predicate `challenge(5272117)`. According to the algorithm described in 2.1, the definition of `valid_credential(studentcard, hu)` (i.e. `[r12]`) should be scanned as well. After unification, `[r12]`'s body contains the provisional predicates `public_key(hu, K)` and `verify_signature(studentcard, K)`. Other provisional predicates may be retrieved by inspecting `[r2]` and `[r4]` in the same way, but it will turn out that, according to the metapolicy, just `challenge(5272117)` and `public_key(hu, K)`

- have `self` as actor
- were not evaluated yet and
- can be immediately evaluated

Therefore just these two will be executed.

Extracting actions (Second iteration) Let suppose that `challenge(5272117)` was successfully executed and that the successful execution of `public_key(hu,`

K) provided the unification K/2172705. The second scan of the policy, performed like the first one, will retrieve the predicate verify_signature(studentcard, 2172705) ready to be executed.

Extracting actions (Third iteration) Let suppose that verify_signature(studentcard, 2172705) was successfully executed. The third scan of the policy will retrieve no provisional predicates, therefore the second step of the algorithm described in 2.1 will be performed.

Proof The proof of the peer request will be performed like a normal Prolog proof, it is hence easy to see that it succeeds.

Filtering The peer request matches [r1], [r2] and [r4]. After unification with the credential sent by the peer, [r1] looks like this.

```
[r1]allow(access(books)) :-
   credential(studentcard[type:student, issuer:hu, public_key: 5272117]),
   valid_credential(studentcard, hu),
   recognized_university(hu),
   challenge(5272117).
```

According to the algorithm described in 2.3, provisional predicates credential(...) and challenge(5272117) are filtered out and the filtered version of [r1] (to be sent to the other peer) looks like this.

```
[r1]allow(access(books)) :-
   valid_credential(studentcard, hu),
   recognized_university(hu).
```

Now [r1]'s body contains the public abbreviation predicate valid_credential(studentcard, hu) and the public state query predicate recognized_university(hu) whose definitions, according to the algorithm described in 2.3, should be scanned as well. After unification [r12] looks like this.

```
[r12]valid_credential(studentcard, hu) :-
   public_key(hu, 2172705),
   verify_signature(studentcard, 2172705).
```

[r12]'s body consists of two provisional predicates which, according to the algorithm described in 2.3, should be filtered out. Therefore the filtered version of [r12] is the fact

```
[r12]valid_credential(studentcard, hu).
```

[f2] is a fact, therefore its body is empty and does not need to be scanned. [f2] is simply added to the filtered policy. [r2] and [r4] can then be filtered in the same way.

As we can see, the filtered policy we obtained by exploiting our filtering procedure is like the one presented in [1] after the 7th step (pg. 51), except for the point described in Note 3.

References

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

[2] L. Sterling, E. Shapiro (1986) *The Art of Prolog*