Protune Prolog interface

J. L. De Coi

September 15, 2006

Abstract

The `org.protune.core.Status` interface provides the following methods allowing to add to the state notifications (resp. checks, filtered policies). On the Prolog side these entities must be somehow coded. Moreover the `Status` interface provides a number of query methods allowing to retrieve the current negotiation step number and, for each previous negotiation step, the notifications (resp. checks, filtered policies) inserted in the state during that negotiation step. On the Prolog side these methods correspond to goals to be satisfied. In the following a suggested encoding as well as some relevant queries are provided.

1 Suggested encoding

1.1 Actions

According to the `javadoc` documentation, an `org.protune.api.Action` has exactly two subclasses `org.protune.api.ReadAction` and `org.protune.api.SideEffectAction`, which differ syntactically only under this point: `ReadAction` returns a result whereas `SideEffectAction` does not return anything. Therefore they could be mapped in Prolog as follows

- `<readActionName>(argumentList, Result)`
- `<sideEffectActionName>(argumentList)`

or as follows

- `<readActionName>(argument1, ..., argumentN, Result)`
- `<sideEffectActionName>(argument1, ..., argumentN)`

the first encoding being clearer under the conceptual point of view, the second one under the practical one.

In Tab. 1 the actions (state predicates) listed in deliverable `I2-D2` are rewritten according to these two syntaxes.[1] The encodings are sligthly different from the one presented in deliverable `I2-D2`, where some syntactical sugar was added.

[1] For the semantics of the predicates, see the `javadoc` documentation.

request([n, R], Result)	request(n, R, Result)
in([packageName, function, <argList>], Result)	in(packageName, function, <argList>, Result)
time(Result)	time(Result)
credential([C, K])	credential(C, K)
declaration([D])	declaration(D)
do([uriOrServiceRequest])	do(uriOrServiceRequest)
authenticatesTo([K])	authenticatesTo(K)
logged([X, logfileName])	logged(X, logfileName)

Table 1: Encodings for actions (state predicates) listed in deliverable I2-D2.

Note: For each parameter of an action and for the return value, suitable Prolog encodings should be found.

1.2 Notifications

Each time an action is performed, a notification of the execution is returned. In the current java implementation, a notification contains a link to the performed action. I claim that such a reference should be present also in the Prolog encoding, which could be something like this.

$$< notificationName > (< action >)$$

In the following two examples are provided.

- `actionWellPerformed(time(Result))`
- `actionWrongPerformed(logged(X, logfileName))`

More comfortable names can be chosen in the future.
So far `org.protune.api.ActionWellPerformed` and `org.protune.api.ActionWrongPerformed` are the only subclasses of `org.protune.api.Notification`. See `javadoc` documentation for further details.

1.3 Checks

Each time a Peer receives a notification, it tries to check whether it is (not) reliable. According to the result of the checking process, a `Check` object is returned. In the current java implementation a check contains a link to the checked notification. I claim that such a reference should be present also in the Prolog encoding, which could be something like this.

$$< checkName > (< notification >)$$

In the following two examples are provided.

- `notificationReliable(actionWellPerformed(time(Result)))`

- `notificationUnreliable(actionWrongPerformed(logged(X, logfileName)))`

More comfortable names can be chosen in the future.
So far `org.protune.api.NotificationReliable` and `org.protune.api.NotificationUnreliable` are the only subclasses of `org.protune.api.Check`. See `javadoc` documentation for further details.

1.4 Filtered policies

At each negotiation step a peer sends/receives one or more filtered policies, which are inserted into the negotiation state as well, therefore a suitable Prolog encoding for filtered policies need to be found. In the following we will assume that a filtered policy is nothing else than a Prolog theory (which strictly speaking is not true: e.g. the term *blurred* is not a keyword in Prolog whereas in Protune it is). According to the previous convention

- a filtered policy is a list of clauses
- a clause is a pair $(head, body)$
- a head is a literal
- a body is a list of literals

Therefore a filtered policy could be represented as follows.

```
[
  [
    literal_1,
    [ literal_11, ... literal_1n_1 ]
  ],
  ...
  [
    literal_m,
    [ literal_m1, ... literal_mn_m ]
  ]
]
```

1.5 Negotiation step elements

As explained before, a negotiation state can contain (among other)

- notifications
- checks
- filtered policies

It could be the case that additional information is required, which is not contained in these entities *per se*. In order to allow more powerful reasoning capabilities could be valuable for each entity tracking

- whether the entity was sent by the current peer to the other one or the other way around
- when the entity was sent/received or when it was inserted into the state
- during which negotiation step the entity was exchanged
- ...[2]

Therefore it could be valuable defining an entity enclosing such information, something like this.[3]

$negotiationElement(negotiationStep, timestamp, sentOrReceived, < entity >)$

The

- `negotiationStep`
- `timestamp`
- `sendOrReceived` flag

could be encoded

- as an integer (eventually starting with 0)
- in a Java-like way (as the number of milliseconds spent since January 1, 1970, 00:00:00 GMT)
- as a boolean (let us say `false` if received and `true` if sent)
- ...[4]

In the following two examples are provided (in the second one the `negotiationID` is present as well).

- `negotiationElement(0, 1000000000, 0, actionWellPerformed(time(Result)))`
- `negotiationElement(1979, 111, 9876543210, 1, notificationUnreliable(actionWrongPerformed(logged(X, logfileName))))`

[2] In case the inference engine provides support for concurrent negotiations, an identifier of the current negotiation should belong to this kind of information as well.

[3] Resp. $negotiationElement(negotiationID, negotiationStep, timestamp, sentOrReceived, < entity >)$

[4] The `negotiationID` could be encoded as an increasing integer.

2 Relevant queries

Let us think that our inference engine does not support concurrent negotiations: therefore at each moment the state contains (among other) a number of statements (i.e. Prolog facts) like these ones.

```
negotiationElement(1, 1000000000, 0, actionWellPerformed(time(Result))).
...
negotiationElement(10, 100000000, 1,
   notificationUnreliable(actionWrongPerformed(logged(X, logfileName)))).
...
negotiationElement(100, 10000000, 0, [ [ allow(release(credential)), [] ] ]).
...
```

By exploiting the reasoning capabilities of Prolog, retrieving e.g. all filtered policies exchanged during a negotiation step or all filtered policies received during the whole negotiation is almost straightforward. Have a look at the following examples.

- $exchangedFilteredPolicies(negotiationStep, L) :-$
  $findall(FilteredPolicy, negotiationElement(negotiationStep, \_, \_, FilteredPolicy), L).$
- $receivedFilteredPolicies(L) :-$
  $findall(FilteredPolicy, negotiationElement(\_, \_, 0, FilteredPolicy), L).$

List L contains the requested filtered policies. In the first example *negotiationStep* represents the number of the negotiation step, the filtered policies exchanged during which are looked for.

Retrieving negotiations or checks is almost as easy as retrieving filtered policies, as the following examples show.

-
-

In the first example all notifications of type `ActionWrongPerformed` exchanged during the negotiation step `negotiationStep` are retrieved. In the second one all checks of type `NotificationReliable` received during the whole negotiation are retrieved.

It is worth mentioning that, in order to retrieve each type of negotiations/checks, the inference engine *should be aware of the currently available types*. Rules targeted to our system have the following form.

$exchangedNegotiations() :- cat(L1, L2), .$