



I2-D16

Integrating Legacy Systems in Protune

Project number:	IST-2004-506779
Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	CO (confidential, only for REWERSE partners)
Document number:	IST506779/L3S Research Center/I2-D16/D/CO/a0.0
Responsible editor(s):	Daniel Olmedilla
Contributing participants:	Hannover, Naples, Munich
Contributing workpackages:	I2,I4
Contractual date of delivery:	No apply
Actual date of delivery:	No apply

Abstract

Policy languages and frameworks focus on the specification of statements defining the behaviour a system must follow. They typically assume that the knowledge base containing those policies are the main source of information. However, companies and institutions may require the use of legacy systems in order to avoid a costly and complex migration. Furthermore, due to performance and scalability issues, it even may not be appropriate to migrate all the data of a company to the policy KB. This document describes scenarios in which legacy systems are or may be used, provides a common interface in PROTUNE for their integration, and enumerates the adequate wrappers required to integrate the appropriate backend repositories.

Keyword List

Policy, wrappers, backend repository, legacy systems, information sources, data integration

Integrating Legacy Systems in Protune

P. A. Bonatti¹, C. Duma², T. Furche³, and D. Olmedilla⁴

¹ Università di Napoli Federico II
Email: bonatti@na.infn.it

² Department of Computer and Information Science, Linköpings universitet
Email: cladu@ida.liu.se

³ Ludwig-Maximilians-Universität München
Email: Tim.Furche@ifi.lmu.de

⁴ L3S Research Center and Hannover University
Email: olmedilla@l3s.de

28 April 2006

Abstract

Policy languages and frameworks focus on the specification of statements defining the behaviour a system must follow. They typically assume that the knowledge base containing those policies are the main source of information. However, companies and institutions may require the use of legacy systems in order to avoid a costly and complex migration. Furthermore, due to performance and scalability issues, it even may not be appropriate to migrate all the data of a company to the policy KB. This document describes scenarios in which legacy systems are or may be used, provides a common interface in PROTUNE for their integration, and enumerates the adequate wrappers required to integrate the appropriate backend repositories.

Keyword List

Policy, wrappers, backend repository, legacy systems, information sources, data integration

Contents

1	Introduction	1
2	Protune language	1
3	Scenarios	4
3.1	RDBMS, XML and RDF repositories	5
3.2	Access to file system	6
3.3	Processing files	6
3.4	E-mail databases	6
3.5	Reputation and recommendation packages	6
4	Interface	8
5	Wrappers	11
5.1	Xcerpt	11
5.2	JDBC	11
A	Protune grammar	12
B	WSDL Interface	14

1 Introduction

Policy languages and frameworks focus on the specification of statements defining the behaviour a system must follow. They typically assume that the knowledge base containing those policies are the main source of information. However, companies and institutions may require the use of legacy systems in order to avoid a costly and complex migration. For instance, existing relational databases may contain information about thousands of clients and resources, or RDF repositories metadata to be used on policy decisions. In order to integrate all this already available information, it should not be required its migration to the knowledge base and language which contains the policies. . Furthermore, due to performance and scalability issues, it even may not be appropriate to migrate all the data of a company to the policy KB. Instead, it should be possible to import dynamically the relevant information required by policies from the sources where it is stored. Integration of legacy systems has been already proven to be required for imperative programming languages like Java or C and policy languages are not an exception.

The PROTUNE policy language allows the import of legacy data by means of package calls and aims at an easy integration with other sources of information. This document describes scenarios in which legacy systems are or may be used, provides a common interface in PROTUNE for their integration, and enumerate the adequate wrappers required to integrate the appropriate backend repositories.

2 Protune language

The PROTUNE rule language is based on normal logic program rules

$$A \leftarrow L_1, \dots, L_n \quad (1)$$

where A is a standard logical atom (called the *head* of the rule) and L_1, \dots, L_n (the *body* of the rule) are literals, that is, L_i equals either B_i or $\neg B_i$, for some logical atom B_i .

A *policy* is a set of rules shaped like (1), such that negation is applied neither to *provisional predicates* (defined below), nor to any predicate occurring in a rule head. This restriction ensures that policies are *monotonic* in the sense of [Seamons et al., 2002], that is, as more credentials are released and more actions executed, the set of permissions does not decrease. Moreover, the restriction on negation makes policies *stratified programs*; therefore negation as failure has a clear, PTIME computable semantics that can be equivalently formulated as the perfect model semantics, the well-founded semantics or the stable model semantics [Baral, 2003].

The vocabulary of predicates occurring in the rules is partitioned into the following categories:

- *Decision predicates*: Currently this class comprises predicates `allow` and `sign`. These predicates are defined in the policy, that is, they occur in the head of some policy rules.

The unary predicate `allow` is queried by the negotiator for access control decisions. The argument of `allow` can denote a service call (for access control decisions) or it can be `release(credential)` or `execute(action)` (for privacy protection). In response to a service request s , the negotiator looks for a (partial) proof of `allow(s)`, and handles it as sketched in the previous section. Similarly, in response to a credential request or an action request r , the negotiator looks for a proof of `allow(r)` and processes it appropriately.

Predicate `sign` is used to issue statements signed by the principal owning the policy. The argument of `sign` can be any term, possibly consisting of attribute-value pairs. This feature is useful to issue new credentials stating domain-dependent properties. In response to a statement request r , a (partial) proof of `sign(r)` is searched for.

- *Abbreviation/abstraction predicates*: These are predicates defined in the policy. They have many purposes ranging from the definition of high-level client properties (e.g. by combining low-level data and/or different credentials, cf. [Bonatti and Samarati, 2000]) to the specification of new credential semantics.
- *Constraint predicates* comprise the usual equality and disequality predicates.
- *State predicates*: Policy decisions have to be taken with respect to a time-dependent system *state*, encoding the current negotiation state, legacy data, user profiles, and so on. State predicates are further partitioned into the following subclasses.
 - *State query predicates*: These predicates read the current state without modifying it. They comprise both built-in and application dependent predicates. Built-in state predicates model the state of the negotiation, and provide a uniform interface to external packages in the style of HERMES [Subrahmanian et al., 1995]. An example of negotiation state atom is `request(n, R)`; it holds if R is the n -th request in the negotiation. External packages (including databases and other data sources) can be queried with atoms of the form:

$$\text{in}(X, \text{package_name} : \text{function}(\text{arg_list})) \quad (2)$$

where the variable X ¹ ranges over the set of objects returned by the code call `package_name : function(arg_list)`. For example if the code call is

```
access : query('select T where A = c'),
```

then conceptually speaking the local state contains all the ground instances of (2) such that X is bound to a tuple of table T with attribute $A = c$. In practice, the implementation needs suitable wrappers for the packages and appropriate solution caching techniques.

- *Provisional predicates*: These are predicates that may be made true by means of appropriate *actions* that may modify the current state. Such actions may be carried out by the server, by the client, or both.

An important example is `credential`. An atom `credential(C, K)` is true when the current negotiation state contains a verified credential matching C and signed by the principal whose public key is K . If this condition is not satisfied, still (an instance of) `credential(C, K)` can be made true by searching for the credential (either directly or by asking the peer to provide it) and loading it into the negotiation state after verification.

Similarly, the `declaration` predicate is satisfied if the peer releases a declaration matching the predicate arguments. The `declaration` predicate is generalized by

¹Note that X should always consist of a predicate name and possibly a set of arguments (variables to be instantiated or constants)

the `do` predicate. Intuitively, `do(uri_or_service_request)` can be made true if the peer connects to `uri` or invokes `service_request`, and then carries out some application dependent procedure. If the procedure is successfully completed, then the atom `do(uri_or_service_request)` becomes true in the negotiation state.

By means of another kind of built-in provisional atoms, `authenticates_to(K)`, the peer can be asked to prove to be the owner of the private key associated to `K`, through a standard challenge procedure.

Sometimes, the actions associated to provisional predicates are to be executed locally, by the negotiator. A common example is `logged(X, logfile_name)` that may be made true by recording `X` into `logfile_name`. The following sample rule `R` records in `ac.log` that access to service `Srv` has been granted by `R` itself:

```
allow(Srv) ←
    ..., logged(Srv + 'granted by R', ac.log).
```

In order to prove `allow(Srv)` from the client's credentials, the system can write the logfile, thereby triggering the rule.

Provisional predicates may be used to encode business rules. For instance, the next rule enables discounts on low_selling articles in a specific session:

```
allow(Srv) ← ..., session(ID),
    in(tuple(X), sql:query('select * from low_selling')),
    enabled(discount(X), ID).
```

Intuitively, if `enabled(discount(X), ID)` is not yet true but the other conditions are verified, then the negotiator may execute the action associated to `enabled` and the rule becomes applicable (if `enabled(discount(X), ID)` is already true, no action is executed). The action associated to `enable` in this case is application dependent. In the next section we shall see how to define such application-specific provisional predicates.

Sometimes actions should be executed *before* asking the peer for credentials. In the next rule the log action is meant to record the incoming request, and must be executed immediately and independently from the peer's response. Predicate `time` is a state query predicate, while `unlogged.allow` is an abbreviation predicate, encoding the actual access control decision for service `Srv`:

```
allow(Srv) ← time(T),
    logged(Srv + 'requested at ' + T, req.log),
    unlogged.allow(Srv).
```

Remark 1 *For simplicity, we assume that provisional atoms are orthogonal, in the sense that the action associated to any ground atom `A` cannot change the truth value of any other ground provisional atom.*

The rule language supports object-oriented dot syntax that, however, is only an abbreviation for standard first-order syntax. One can express by `X.attr : v` the fact that `X` has an attribute

`attr` with value v . Actually, $X.attr : v$ abbreviates the standard atom $attr(X, v)$. This representation allows multi-valued attributes. This attribute semantics is compatible with semantic web standards such as RDF and OWL (in particular $X.attr : v$ corresponds to an RDF triple).

More generally, $X.a_1 \dots a_n : v$ abbreviates

$$a_1(X, V_1), a_2(V_1, V_2), \dots, a_n(V_{n-1}, v) \quad (3)$$

where V_1, \dots, V_{n-1} are fresh variables (not used elsewhere) that are meant to be existentially quantified. In practice, this means that when $X.a_1 \dots a_n : v$ occurs in a rule body it abbreviates exactly (3), while asserting $X.a_1 \dots a_n : v$ as a fact causes the atoms in (3) to be asserted as individual facts after replacing V_1, \dots, V_{n-1} with $n - 1$ Skolem constants.

Finally, an atom $A = p(\dots, X.path:v, \dots)$ is expanded to two atoms $p(\dots, X, \dots)$, $X.path:v$. If A occurs in the body of a rule R , then it suffices to expand A in the body of R . If A is the head of R , then the first atom in the expansion is the actual head and the second atom is to be inserted in the body:

$$p(X.path : v) \leftarrow Body \text{ abbreviates } p(X) \leftarrow X.path : v, Body.$$

3 Scenarios

As stated in the language definition, PROTUNE already incorporates a construction that allows using libraries:

```
in( $X$ , package_name : function(arg_list))
```

where the variable X ranges over the set of objects returned by the code call *package_name* : *function*(*arg_list*).

Intuitively, the result of the evaluation of the function will provide the insertion in the knowledge base of the predicate *in* with the objects returned and the call as arguments. For example, assume that a call

```
in(user( $X$ ), rbms : query('select name from users', 'db_users'))
```

is included in our policy and suppose the table *users* in database *db_users* contains the following records:

UserId	Name	Address	Gender
1	Piero Bonatti	Via Mateo, Naples	Male
2	Daniel Olmedilla	Gluenderstrasse, Hannover	Male

In such a case, the evaluation of the policy will produce the following two new facts in the knowledge base:

```
in(user('Piero Bonatti'), rbms : query('select name from users', 'db_users')).
in(user('Daniel Olmedilla'), rbms : query('select name from users', 'db_users')).
```

It is important to note that the return value of the function will be a multiset of variable bindings, that is, repetitions of tuples are allowed. The package implementations may include functions or constructions to remove duplicates (like in SQL using the keyword “distinct”). In addition, there are two reasons why the objects are introduced in the KB as arguments together with the function call:

- We want to distinguish those facts introduced manually by an administrator from those introduced as a result of the execution of a package call
- The function call acts as a context. This way it is possible to distinguish the results of the following two calls

```
in(book(X), rdbms : query('select title from books where author = "Shakespeare"', 'db_books'))
in(book(X), rdbms : query('select title from books where author = "Lorca"', 'db_books'))
```

where otherwise, facts like “book(Title)” would not be bound to any context.

Finally, “null” values are allowed and therefore it might be possible to have in the KB facts like

```
in(user('Alice', nil), rdbms : query('select name, address from users', 'db_users'))
```

In the rest of this section we describe some useful scenarios in which external calls might be used in order to increase the expressiveness and enhance the evaluation process.

3.1 RDBMS, XML and RDF repositories

Policies may want to use data that already exist in a company like which kind of subscription a client has or whether she is a gold member. Some examples with the appropriate policy are:

- *Retrieving from a database the subscription of a client:* A client may access a document if her subscription includes that resource:

```
allow(download(Document)) ←
  authenticate(User),
  in(has_subs(User, Subs), rdbms : query(
    'select user, subscription from users where user = ' + User, 'db_users'
  )),
  includedIn(Document, Subs).
```

- *Checking if a client is a VIP member:* A client may access a document if she is a gold client:

```
allow(download(Document)) ←
  authenticate(User),
  tobedonebyTimwithaXQueryorXcerptquery : -),
  includedIn(Document, Subs).
```

- *Querying metadata related to a decision:* Any author of a document may access it:

```
allow(download(Document)) ←
  authenticate(User),
  in(dc:Author(Document, Author), rdf : RQLquery(
    PREFIX dc : < http : // purl . org / dc / elements / 1.1 / >
    'SELECT ?title WHERE ?title dc : author ?' + User
  )).
```

3.2 Access to file system

Typically, registering at a company includes accepting a “terms and license agreement”. The policy in charge of providing the appropriate text may have the agreement hardcoded or, in a more convenient way, may retrieve it from a text file stored somewhere in the file system.

3.3 Processing files

Some policies may want to protect information based on the content. Some companies may decide to have a denial constraint stating that no document is given access if it contains the word “confidential” or “draft”. Another example may be policies in which parents state that any webpage where the words “sex” or “porn” appears should be forbidden for anyone using the computer.

3.4 E-mail databases

Current authentication mechanisms include validation of e-mails. The process is simple, an e-mail is sent to the user with a random generated number and the client is asked to reply to that one. Therefore, it is possible to include in a policy a condition which will allow the authentication of a user only if she has already validated her e-mail. That requires a query to the server e-mail database in order to confirm that such an e-mail with the appropriate random generated string and coming from the provided e-mail has been received.

3.5 Reputation and recommendation packages

Transaction policies must handle expenses of all magnitudes, from micropayments (e.g. a few cents for a song downloaded to your iPod) to credit card payments of a thousand euros (e.g. for a plane ticket) or even more [Bonatti et al., 2005]. The cost of the traded goods or services typically contributes to determining the risk associated to the transaction and hence the trust needed for performing it. For instance, for micro-payments of a few euros or cents, a seller could just check the reputation of the buyer within the community. If the buyer’s reputation is high, the risk that he or she would not pay is very low, and thus the transaction can be conducted with a simple check. On the contrary, if a buyer’s reputation is low or the amount of money involved in the transaction is high, risk is higher and thus the seller may require stronger guarantees, such as a verified credit card number to ensure that the buyer can and will pay.

The buyer’s point of view is dual. If the amount of the transaction is high, the buyer may require strong and objective guarantees that the seller will deliver the goods and that the credit card will not be misused. For example, the buyer may require a secure connection, BBB (Better Business Bureau) certificates, blacklist checks, etc. In addition, the buyer may consider the seller’s reputation in the community to increase the chances of successful transaction completion and privacy protection.

This might be of particularly interest when many parties provide the same or similar services, and the experience of others in the community could play an important role in the final provider selection.

To acquire the needed trust might be tedious and dependent on an infrastructure (e.g. secure connection with the card issuer, certification authorities, etc). But such an infrastructure might not be always available. For instance, with the advent of P2P and the Semantic Web, where each participant could be provider and a consumer, it is difficult to assume that all the participants

will be registered with certification authorities. Moreover, many times, the new markets are of low value. On the other hand, for small transactions, such as buying a new song, the risk is low and so could be the required guarantees. In this case, the user might just check the reputation of

In E-Bay, each buyer and seller can check whether the counterpart (seller and buyer) is trustworthy based on community feedback and also according to policies (if the price/risk is high, then more trust is required).

In order to implement such a scenario, it is required to integrate reputation mechanisms into policies. Reputation-based trust can be formalized by relations between *trustors*, *trustees*, *actions*, and *trust levels* [Staab et al., 2004]. For instance, a fact like

```
trust(P, S, diagnosis(viral), 80–100)
```

would model the fact that patient P trusts specialist S on diagnosis of viral diseases with an estimated confidence level belonging to the interval $80 - 100$.

Such trust statements can be the basis for trust propagation (e.g. via rules such as “trust X as a bike mechanic if X is trusted as a car mechanic”), for access control decisions such as:

```
allow(download(contents/pre_release))
← user(X),
  trust(self, X, download(contents/pre_release), 90–100).
```

as well as *recommendations*, that is, statements like

```
recommend(AmbulanceSupervisor, _Paramedic, JoinResponseTeam, high)
← employed(LondonAmbulance, _Paramedic).
```

Such decisions may consider a notion of *risk*, as in

```
trust(ProgramX, Server, storeData(Server), 80–100)
← Server.owner:CoXYZ,
  risk(fail(Server), 0–0.1).
```

These examples (taken from [Staab et al., 2004]) show how trust and recommendations can be modelled and applied through a small set of predicates. The problem is: How should the basic *facts* about trust and risk be gathered and maintained?

In some case, such facts can be defined by standard policy rules, for example:

```
trust(A, B, download(file), 80–100) ← credential(X, VISA),
                                       X.type : credit_card,
                                       X.owner : B.
```

However, the main current approaches are based on numerical models (see [Bonatti et al., 2004] for an extensive illustration of the main approaches) and ad-hoc algorithms for gathering, processing, and propagating historical data about past interactions and the resulting trust measures. In perspective, it may be possible to apply probabilistic, possibilistic or annotated logics to handle such numbers, but so far there is no clear indication that this is the right direction, nor any hint on how to do it.

Further difficulties are: (i) data are application dependent, as well as the procedures for obtaining them; (ii) trust is a dynamic concept, i.e., it changes over time.

The above difficulties suggest a modular approach, namely, the computation and distribution of the basic facts on reputation and risk are delegated to suitable external packages. The results of their processing can be imported via HERMES-like state predicates such as

```
in(trust(X, Y, A, L), reputation_pckg : eval_trust())
```

```
in(risk(X, L), reputation_pckg : eval_risk())
```

(cf. (2)). In the above examples the functions `eval_trust()` and `eval_risk()` wrap queries to the underlying reputation management and risk assessment algorithms, whatever they are. The two wrappers collect and return the results of those subsystems as a set of terms matching the first argument of the `in` predicate. Then non-rule-based reputation and risk models can be integrated in PROTUNE policies without any ad-hoc language primitives.

Another advantage of this approach is that a single policy may simultaneously apply different approaches to reputation simply by invoking different packages and combining their results with suitable rules. This kind of flexibility is particularly important in a stage where it is not yet clear which of the competing models of reputation-based trust will become widely accepted, and which application domains they will prove to be good for. It is also possible to change the number and type of parameters of the `trust` and `risk` predicates, if needed by a particular reputation model.

This flexible architecture is compatible both with *on-demand* trust computation and with proactive propagation of trust evaluation, as reputation packages may receive asynchronous messages from other peers, concerning warnings and reputation evaluations.

4 Interface

In previous sections we have described how the integration of library functions is performed within the PROTUNE language. While this information is sufficient for policy administrators and users, developers willing to integrate their own libraries will require a detailed description of the architecture used in our implementation. This section aims at providing developers such information.

Figure 1 depicts the high level architecture of a PROTUNE Agent architecture. It is composed by the following modules:

- *Inference Engine*. Loads at startup the policies that are going to be used as well as possibly some metadata information about the resources to be protected and relevant ontologies. It is in charge of performing the policy filtering and evaluation.
- *Credential Repository*. It is in charge of loading the local credentials, feeding the inference engine with the representation of those credentials in order to be used during the policy evaluation and the negotiation module when disclosure of a credential is required.
- *Negotiation Module*. Guides the negotiation and manage the information flow between modules.
- *Strategy Evaluator*. It is in charge of applying a relevant strategy during negotiations.

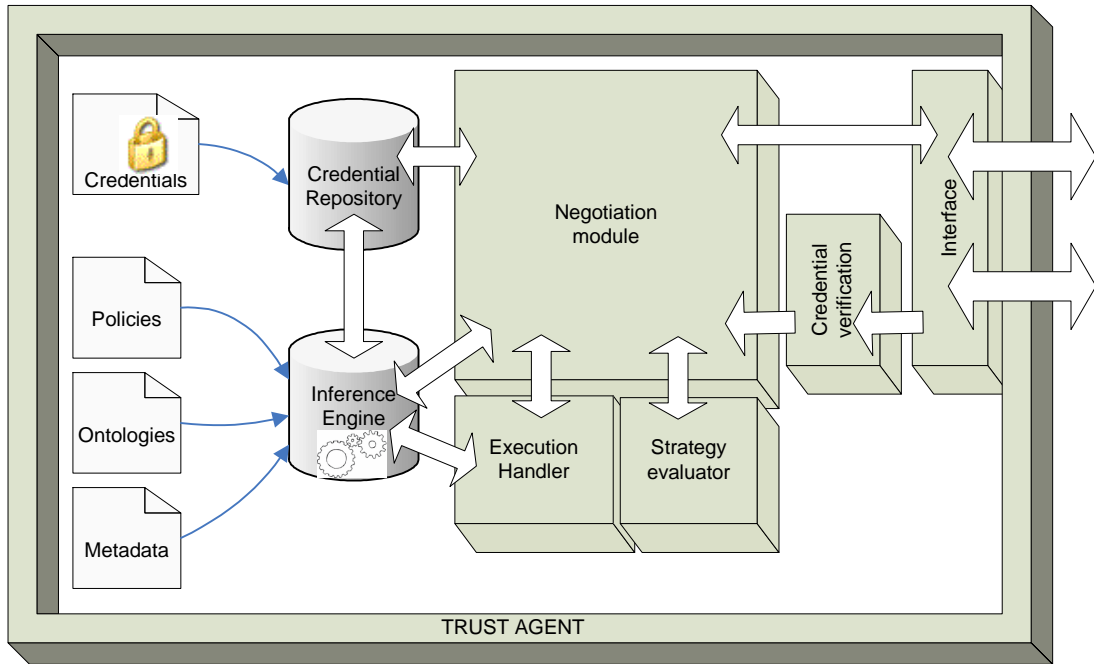


Figure 1: PROTUNE Agent Architecture

- *Execution Handler*. Responsible for executing actions and package calls specified in the policies and feeding the inference engine with the result of the executions.
- *Credential Verification Module*. When receiving a new credential from other party, this module is responsible for checking that the credential is not forged, that is, the signature and Certification Authority are valid.
- *Communication Interface*. It is in charge of the communication with other parties. Some example of possible interfaces are secure socket connections or web services.

In order to facilitate the use of new functions in libraries, we adopt the following interface (see WSDL file in appendix B) which will have to be adopted by developers:

<i>Method Name</i>	executeAction	
<i>Return Type</i>	ActionResult	
<i>Parameters</i>	<i>Name</i>	<i>Type</i>
	functionName	String
	arguments	String
	inputVars	String
<i>Fault</i>	NO_SUCH_FUNCTION INVALID_ARGUMENT FUNCTION_FAILURE	

where Action The argument *functionName* contains the name of the function in the package, *arguments* is an array of strings containing the list of arguments in a sequence (in order to

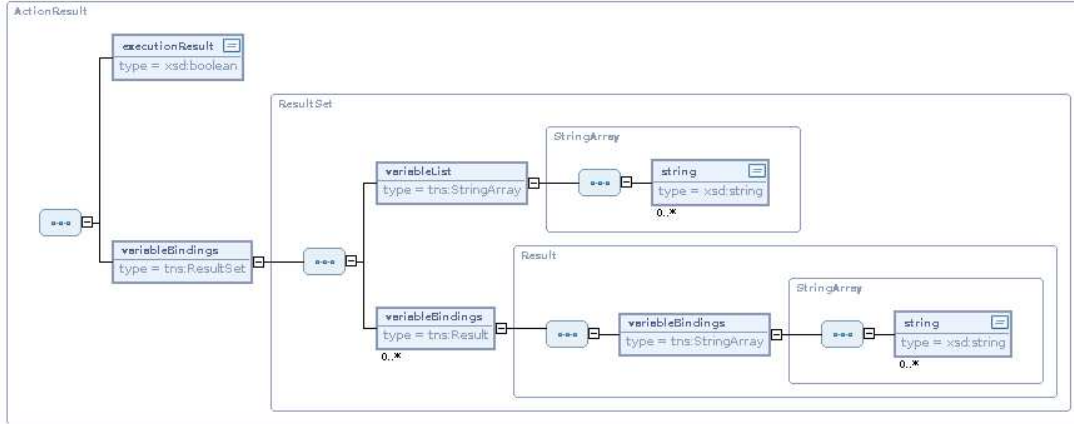


Figure 2: ActionResult Datatype description

allow multiple arity) and *inputVars* provides an array of strings with the variables for which the requester wants to receive a binding. The result is given by the return value: an Action Result (see figure 2). It contains a boolean value, either true or false, describing whether the query was succesfull (even if no bindings are returned), and *ResultSet*, which is again a complex datatype and contains the variable bindings. A *ResultSet* contains the list of variables (in order to make it self-contained) and an array of results (each one representing a row in the resultset².

We have decided to follow a similar interface as the one other prolog interfaces provide. On the one hand, the variable bindings is required in order to provide with the query results (as in the first example in section 3). On the other hand, in some cases, it might be needed to check just the truth value” of a query in which case no variable binding is returned. For example, suppose we have a policy like

```
validUser(User) ←
  credential(C),
  C.name : User,
  in(user(User), rdbms : query('select name from users', 'db_users'))
```

and a the requester, “Alice”, has already submitted a credential together with her request. In such a case, the variable “User” is already instantiated when executing the rdbms query and therefore, the query sent to the execution handler would be

```
in(user('Alice'), rdbms : query('select name from users', 'db_users'))
```

where no variables are passed as arguments. In such a case, an empty resultset could be considered either as “there is a user Alice in the database but no variable bindings returned” (since there were not variables specified in the request) or “there is no user Alice in the database and therefore no variable bindings are returned”. In order to disambiguate, a boolean is used in order to inform the execution handler of whether the above fact should be returned to the knowledge base or not.

²While currently we do not enforce the return of type information in the resultset, it might be useful to do so in the future, when PROTUNE is typed

In addition, the following faults are defined in the interface:

- *NO_SUCH_FUNCTION*: produced if the function specified as argument is not a valid function name or is not implemented.
- *INVALID_ARGUMENT*: raised in case any of the arguments in the comma separated list is not valid.
- *FUNCTION_FAILURE*: fault that should catch internal errors in the execution of the function.

All the previously faults should be accompanied, in case of being raised, of an explanation message to be displayed in order to provide more information regarding to the reason that produced such a fault.

Finally, the *execution handler* is also in charge of resolving the package name and assigning it/making a call to the right implementation. Furthermore, it must also feed the inference engine with the results of the function execution.

5 Wrappers

This section describes the different packages integrated (or to be integrated) within the PRO-TUNE framework, the functions provided and some examples of its use.

5.1 Xcerpt

5.2 JDBC

References

- [Baral, 2003] Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, Cambridge.
- [Bonatti and Samarati, 2000] Bonatti, P. and Samarati, P. (2000). Regulating service access and information release on the web. In *CCS '00: Proceedings of the 7th ACM conference on computer and communications security*, pages 134–143. ACM Press.
- [Bonatti et al., 2005] Bonatti, P. A., Duma, C., Olmedilla, D., and Shahmehri, N. (2005). An integration of reputation-based and policy-based trust management. In *Semantic Web Policy Workshop in conjunction with 4th International Semantic Web Conference*, Galway, Ireland.
- [Bonatti et al., 2004] Bonatti, P. A., Shahmehri, N., Duma, C., Olmedilla, D., Nejd, W., Baldoni, M., Baroglio, C., Martelli, A., Patti, V., Coraggio, P., Antoniou, G., Peer, J., and Fuchs, N. E. (2004). Rule-based policy specification: State of the art and future work. Technical report, Working Group I2, EU NoE REVERSE. <http://reverso.net/deliverables/i2-d1.pdf>.
- [Seamons et al., 2002] Seamons, K., Winslett, M., Yu, T., Smith, B., Child, E., Jacobson, J., Mills, H., and Yu, L. (2002). Requirements for policy languages for trust negotiation. In *POLICY '02: Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 68. IEEE Computer Society.

- [Staab et al., 2004] Staab, S., Bhargava, B. K., Lilien, L., Rosenthal, A., Winslett, M., Sloman, M., Dillon, T. S., Chang, E., Hussain, F. K., Nejd, W., Olmedilla, D., and Kashyap, V. (2004). The pudding of trust. *IEEE Intelligent Systems*, 19(5):74–88.
- [Subrahmanian et al., 1995] Subrahmanian, V., Adali, S., Brink, A., Lu, J., Rajput, A., Rogers, T., Ross, R., and Ward, C. (1995). *HERMES: Heterogeneous reasoning and mediator system*. <http://www.cs.umd.edu/projects/hermes>.

A Protune grammar

Not yet the last version.

```

<Attribute>-> <Constant>
<Argument>-> <Term>
| <Package> : <Function> ( <Term> )
<ArgumentList>-> <Argument> <ArgumentList>
| E
<Arguments>-> ( <ArgumentList> )
| E
<Body>-> <RuleSep> <Literal> <LiteralList>
| E
<Constant>-> <StringConstant>
| <Digit> <StringExtended>
| <StringQuoted>
<Digit>-> 0 | 1 | ... | 9
<Directive>-> include <String>
<DirectiveList>-> <Directive> <DirectiveList>
| E
<Fact>-> <Head> <RuleSep>
| <Head>
<Function>-> <StringConstant>
<Head>-> <Literal>
| <NegSymbol> <Literal>
<Id>-> <Constant> :
| E
<Literal>-> <Predicate> <Arguments>
| <Term> <Operator> <Term>
<LiteralList>-> , <Literal> <LiteralList>
| E
<LowerCase>-> a | b | ... | z
<MetaBody>-> <RuleSep> <Literal> <MetaLiteralList>
| <RuleSep> <MetaLiteral> <MetaLiteralList>
| E
<MetaHead>-> <Id>
| <Literal>
| <NegSymbol> <Literal>
<MetaLiteral>-> <Literal> . <Attribute> : <Value>

```

```

| <NegSymbol> <Literal> . <Attribute> : <Value>
<MetaLiteralList>-> , <MetaLiteral> <MetaLiteralList>
| , <Literal> <MetaLiteralList>
| E
<MetaRule>-> <MetaHead> . <Attribute> : <Value> <MetaBody> .
<NegSymbol>-> not
| \=
<Operator>-> =
| >
| >=
| <
| <=
| !=
<Package>-> <StringConstant>
<Predicate>-> <StringConstant>
<Program>-> <DirectiveList> <RuleList>
<Rule>-> <Id> <Head> <Body> .
<RuleList>-> <Rule> <RuleList>
| <MetaRule> <RuleList>
| E
<RuleSep>-> <-
| :-
<String>-> <LowerCase> <String>
| <LowerCase>
| <UpperCase> <String>
| <UpperCase>
| <Digit> <String>
| <Digit>
<StringConstant>-> <LowerCase> <StringExtended>
<StringExtended>-> <String> <StringExtended>
| <String>
| _ <StringExtended>
| -
<StringQuoted>-> " <Text> "
| ' <Text> '
<Term>-> <Variable>
| <Constant>
| [ <TermList> ]
<TermList>-> <Term> , <TermList>
| <Term>
| E
<Text>-> <StringExtended> <Text>
| <StringExtended>
| : <Text>
| :
| / <Text>
| /

```

```

| . <Text>
| .
<Value>-> <Term>
| <URI>
<Variable>-> <UpperCase> <StringExtended>
| _ <StringExtended>
| _
<UpperCase>-> A | B | ... | Z
<URI>-> <Text>

```

B WSDL Interface

```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://action.policy.org/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="ExecutablePackage"
targetNamespace="http://action.policy.org/">

<wsdl:types>
<xsd:schema targetNamespace="http://action.policy.org/">

<xsd:element name="StringArray" type="tns:StringArray"/>
<xsd:complexType name="StringArray">
<xsd:sequence>
<xsd:element name="string"
type="xsd:string" minOccurs="0"
maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>

<xsd:element name="Result" type="tns:Result"/>
<xsd:complexType name="Result">
<xsd:sequence>
<xsd:element name="variableBindings"
type="tns:StringArray" />
</xsd:sequence>
</xsd:complexType>

<xsd:element name="ResultSet" type="tns:ResultSet"/>
<xsd:complexType name="ResultSet">
<xsd:sequence>
<xsd:element name="variableList"
type="tns:StringArray" />
<xsd:element name="variableBindings"
type="tns:Result" minOccurs="0"

```

```

maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>

<xsd:element name="ActionResult"
type="tns:ActionResult"/>
<xsd:complexType name="ActionResult">
<xsd:sequence>
<xsd:element name="executionResult"
type="xsd:boolean" />
<xsd:element name="variableBindings"
type="tns:ResultSet" />
</xsd:sequence>
</xsd:complexType>

<xsd:element name="Exception" type="tns:Exception"/>
<xsd:complexType name="Exception">
<xsd:sequence>
<xsd:element name="text"
type="xsd:string" minOccurs="1" maxOccurs="1" nillable="false"/>
</xsd:sequence>
</xsd:complexType>

<xsd:element name="IllegalArgumentException"
type="tns:IllegalArgumentException"/>
<xsd:complexType name="IllegalArgumentException">
<xsd:complexContent>
<xsd:extension base="tns:Exception">
<xsd:sequence/>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:element name="ActionException"
type="tns:ActionException"/>
<xsd:complexType name="ActionException">
<xsd:complexContent>
<xsd:extension base="tns:Exception">
<xsd:sequence/>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:element name="NoSuchFunctionException"
type="tns:NoSuchFunctionException"/>
<xsd:complexType name="NoSuchFunctionException">
<xsd:complexContent>

```

```

<xsd:extension
base="tns:ActionException">
<xsd:sequence/>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

<xsd:element name="FunctionFailureException"
type="tns:FunctionFailureException"/>
<xsd:complexType name="FunctionFailureException">
<xsd:complexContent>
<xsd:extension
base="tns:ActionException">
<xsd:sequence/>
</xsd:extension>
</xsd:complexContent>
</xsd:complexType>

</xsd:schema>
</wsdl:types>

<wsdl:message name="executeResponse">
<wsdl:part name="executeResponse" element="tns:ActionResult"/>
</wsdl:message>

<wsdl:message name="executeRequest">
<wsdl:part name="function" type="xsd:string"></wsdl:part>
  <wsdl:part name="arguments" element="tns:StringArray"/>
  <wsdl:part name="inputVars" element="tns:StringArray"/>
</wsdl:message>

<wsdl:message name="IllegalArgumentFault">
<wsdl:part name="fault" element="tns:IllegalArgumentException"/>
</wsdl:message>

<wsdl:message name="NoSuchFunctionFault">
<wsdl:part name="fault" element="tns:NoSuchFunctionException"/>
</wsdl:message>

<wsdl:message name="FunctionFailureFault">
<wsdl:part name="fault" element="tns:FunctionFailureException"/>
</wsdl:message>

<wsdl:portType name="ExecutablePackage">
<wsdl:operation name="execute">
<wsdl:input message="tns:executeRequest"/>
<wsdl:output message="tns:executeResponse"/>

```

```

<wsdl:fault name="IllegalArgumentFault"
message="tns:IllegalArgumentFault"/>
<wsdl:fault name="NoSuchFunctionFault"
message="tns:NoSuchFunctionFault"/>
<wsdl:fault name="FunctionFailureFault"
message="tns:FunctionFailureFault"/>
</wsdl:operation>
</wsdl:portType>

<wsdl:binding name="ExecutablePackageSOAP" type="tns:ExecutablePackage">
<soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="execute">
<soap:operation
soapAction="http://action.policy.org/NewOperation"/>
<wsdl:input>
<soap:body parts=" NewOperationRequest"
use="literal"/>
</wsdl:input>
<wsdl:output>
<soap:body parts=" NewOperationResponse"
use="literal"/>
</wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="ExecutablePackage">
<wsdl:port binding="tns:ExecutablePackageSOAP"
name="ExecutablePackageSOAP">
<soap:address location="http://www.example.org"/>
</wsdl:port>
</wsdl:service>

</wsdl:definitions>

```