Protocols

Protocols enable the separation between interface and implementation: several objects can implement the same protocol and an object can implement several protocols. There are no pre-defined protocols in Logtalk.

Defining a new protocol

We can define a new object in the same way we write Prolog code: by using a text editor. Logtalk source files may contain one or more objects, categories, or protocols. If you prefer to define each entity in its own source file, it is recommended that the file be named after the protocol. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the configuration files. Compiled source files (by the Logtalk preprocessor) have, by default, a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding configuration file. For example, we may define a protocol named `listp` and save it in a `listp.lgt` source file that will be compiled to a `listp.pl` Prolog file.

Protocol names must be atoms. Objects, categories and protocols share the same name space: we can not have a protocol with the same name as an object or a category.

Protocol directives are textually encapsulated by using two Logtalk directives: `protocol/1-2` and `end_protocol/0`. The most simple protocol will be one that is self-contained, not depending on any other Logtalk entity:

```
:- protocol(Protocol).
    ...
:- end_protocol.
```

If a protocol extends one or more protocols, then the opening directive will be:

```
:- protocol(Protocol,
    extends(OtherProtocol)).
    ...
:- end_protocol.
```

Finding defined protocols

We can find, by backtracking, all defined protocols by using the `current_protocol/1` built-in predicate with an uninstantiated variable:

```
| ?- current_protocol(Protocol).
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

This predicate can also be used to test if a protocol is defined by calling it with a valid protocol identifier (an atom).

Creating a new protocol in runtime

We can create a new (dynamic) protocol in runtime by calling the Logtalk built-in predicate `create_protocol/3`:

```
| ?- create_protocol(Protocol, Relations, Directives).
```

The first argument, the name of the new protocol (a Prolog atom), should not match an existing entity name. The remaining two arguments correspond to the relations described in the opening protocol directive and to the protocol directives.

For instance, the call:

```
| ?- create_protocol(ppp, [extends(qqq)], [public(foo/1, bar/1)]).
```

is equivalent to compiling and loading the protocol:

```
:- protocol(ppp,
    extends(qqq)).

    :- dynamic.

    :- public(foo/1, bar/1).

:- end_protocol.
```

If we need to create a lot of (dynamic) protocols at runtime, then is best to define a metaclass or a prototype with a predicate that will call this built-in predicate in order to provide more sophisticated behavior.

Abolishing an existing protocol

Dynamic protocols can be abolished using the `abolish_protocol/1` built-in predicate:

```
| ?- abolish_protocol(Protocol).
```

The argument must be an identifier of a defined dynamic protocol, otherwise an error will be thrown.

Protocol directives

Protocol directives are used to set initialization goals and protocol properties.

Protocol initialization

We can define a goal to be executed as soon as a protocol is (compiled and) loaded to memory with the `initialization/1` directive:

```
:- initialization(Goal).
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The argument can be any valid Prolog or Logtalk goal, including a message sending call.

Dynamic protocols

As usually happens with Prolog code, a protocol can be either static or dynamic. A protocol created during the execution of a program is always dynamic. A protocol defined in a file can be either dynamic or static. Dynamic protocols are declared by using the `dynamic/0` directive in the protocol source code:

```
:- dynamic.
```

The directive must precede any predicate directives. Please be aware that using dynamic code implies a performance hit when compared to static code. We should only use dynamic protocols when these need to be abolished during program execution.

Protocol documentation

A protocol can be documented with arbitrary user-defined information by using the `info/1` directive:

```
:- info(List).
```

See the documenting Logtalk programs session for details.

Protocol relationships

Logtalk provides two sets of built-in predicates that enable us to query the system about the possible relationships that a protocol have with other entities.

The built-in predicates `extends_protocol/2` and `extends_protocol/3` return all pairs of protocols so that the first one extends the second:

```
| ?- extends_protocol(Protocol1, Protocol2).
```

or, if we want to know the extension scope:

```
| ?- extends_protocol(Protocol1, Protocol2, Scope).
```

To find which objects or categories implement which protocols we can call the `implements_protocol/2` or `implements_protocol/2` built-in predicates:

```
| ?- implements_protocol(ObjectOrCategory, Protocol).
```

or, if we want to know the implementation scope:

```
| ?- implements_protocol(ObjectOrCategory, Protocol, Scope).
```

Note that, if we use an uninstantiated variable for the first argument, we will need to use the `current_object/1` or `current_category/1` built-in predicates to identify the kind of entity returned.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Protocol properties

We can find the properties of defined protocols by calling the `protocol_property/2` built-in predicate:

```
| ?- protocol_property(Protocol, Property).
```

A protocol may have the property `static`, `dynamic`, or `built_in`. Dynamic protocols can be abolished in runtime by calling the `abolish_protocol/1` built-in predicate.

Implementing protocols

Any number of objects or categories can implement a protocol. The syntax is very simple:

```
:- object(Object,
    implements(Protocol)).
    ...
:- end_object.
```

or, in the case of a category:

```
:- category(Object,
    implements(Protocol)).
    ...
:- end_category.
```

To make all public predicates declared via an implemented protocol protected or to make all public and protected predicates private we prefix the protocol's name with the corresponding keyword. For instance:

```
:- object(Object,
    implements(private::Protocol)).
    ...
:- end_object.
```

or:

```
:- object(Object,
    implements(protected::Protocol)).
    ...
:- end_object.
```

Omitting the scope keyword is equivalent to writing:

```
:- object(Object,
    implements(public::Protocol)).
    ...
:- end_object.
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The same rules applies to protocols implemented by categories.

Copyright © Paulo Moura — Logtalk.org
XHTML + CSS Last updated on: January 21, 2006

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/