Programming in Logtalk

Writing programs

For a successful programming in Logtalk, you need a good working knowledge of Prolog and an understanding of the principles of object-oriented programming. All guidelines for writing good Prolog code apply as well to Logtalk programming. To those guidelines, you should add the basics of good object-oriented design.

One of the advantages of a system like Logtalk is that it enable us to use the currently available object-oriented methodologies, tools, and metrics [Champaux 92] in Prolog programming. That said, writing programs in Logtalk is similar to writing programs in Prolog: we define new predicates describing what is true about our domain objects, about our problem solution. We encapsulate our predicate directives and definitions inside new objects, categories and protocols that we create by hand with a text editor or by using the Logtalk built-in predicates. Some of the information collected during the analysis and design phases can be integrated in the objects, categories and protocols that we define by using the available entity and predicate documenting directives.

Source files

Logtalk source files may contain any number of objects, categories, protocols, and plain Prolog code. If you prefer to define each entity in its own source file, then it is recommended that the source file be named after the entity identifier. For parametric objects, the identifier arity can be appended to the identifier functor. 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 an object named `vehicle` and save it in a `vehicle.lgt` source file that will be compiled to a `vehicle.pl` Prolog file. If we have a `sort(_)` parametric object we can save it on a `sort_1.lgt` source file that will be compiled to a `sort_1.pl` Prolog file. This name scheme helps avoid file name conflicts (remember that all Logtalk entities share the same name space).

Logtalk source files may contain arbitrary Prolog directives and clauses interleaved with Logtalk entity definitions. These directives and clauses will not be compiled by the Logtalk preprocessor and will be copied unchanged to the corresponding Prolog output file. This feature is included to help the integration of Logtalk with other Prolog extensions such as, for example, constraint programming extensions.

Loader utility files

Most examples directories contain a Logtalk utility file that can be used to load all included source files. These loader utility files are usually named `loader.lgt` or contain the word "loader" in their name. Loader files are compiled and loaded like any ordinary Logtalk source file. For an example loader file named `loader.lgt` we would type:

```
| ?- logtalk_load(loader).
```

Usually these files contain a call to the Logtalk built-in predicates `logtalk_load/1` or `logtalk_load/2`, wrapped inside an `initialization/1` directive. For instance, if your code is split in three Logtalk source files named `source1.lgt`, `source2.lgt`, and `source3.lgt`, then the contents of your loader file could be:

```
:- initialization(
    logtalk_load([
        source1, source2, source3])).
```

Another example of directives that are often used in a loader file would be `op/3` directives declaring global operators needed by your application. Loader files are also often used for setting compiler options for the source files (this is useful even when you only have a single source file if you always load it with using the same set of compiler options). For example:

```
:- initialization(
    logtalk_load(
        [source1, source2, source3],
        [underscore_vars(dont_care), portability(warning), xmlspec(xsd)])).
```

To take the best advantage of loader files, assert a clause to the dynamic predicate `logtalk_library_path/2` for the directory containing your source files, as explained in the next section.

A common mistake is to try to set compiler options using `logtalk_load/2` with a loader file. For example, by writing:

```
| ?- logtalk_load(loader, [xmlspec(xsd), xslfile('lgtxhtml.xsl')]).
```

This will not work as you might expect as the compiler options will only be used in the compilation of the `loader.lgt` file itself and will not affect the compilation of files loaded through the `initialization/1` directive contained on the loader file.

Libraries of source files

Logtalk defines a *library* simply as a directory containing source files. Library locations can be specified by asserting clauses to the dynamic predicate `logtalk_library_path/2`. For example:

```
| ?- assertz(logtalk_library_path(shapes, '$LOGTALKUSER/examples/shapes/')).
```

The first argument of the predicate is used as an alias for the path on the second argument. Library aliases may also be used on the second argument. For example:

```
| ?- assertz(logtalk_library_path(lgtuser, '$LOGTALKUSER/')),
     assertz(logtalk_library_path(examples, lgtuser('examples/'))),
     assertz(logtalk_library_path(viewpoints, examples('viewpoints/'))).
```

This allows us to load a library source file without the need to first change the current working directory to the library directory and then back to the original directory. For example, in order to load a `loader.lgt` file, contained in a library named `shapes`, we just need to type:

```
| ?- logtalk_load(viewpoints(loader)).
```

The best way to take advantage of this feature is to load at startup a source file containing an `initialization/1` directive which asserts all the `logtalk_library_path/2` clauses needed for all available libraries. This allows us to load library source files or entire libraries without worrying about libraries paths, improving code portability. The directory paths on the second argument must always end with the path directory separator character.

Unfortunately, a few Prolog compilers do not support the `<library>(<source file>)` notation. In this case, you will need to set the working directory to be the one that contains the source file in order to load it. The library notation provides functionality similar to the `file_search_path/2` mechanism introduced by Quintus Prolog and later adopted by some other Prolog compilers.

Portable programs

Logtalk is compatible with almost all modern Prolog compilers. However, this does not necessarily imply that your Logtalk programs will have the same level of portability. If possible, you should only use in your programs Logtalk built-in predicates and ISO Prolog defined built-in predicates. If you need to use built-in predicates that may not be available in other Prolog compilers, you should try to encapsulate the non-portable code in a small number of objects and provide a portable **interface** for that code through the use of Logtalk protocols. An example will be code that access operating-system specific features. The Logtalk compiler can warn you of the use of non-ISO defined built-in predicates by using the `portability/1` compiler flag.

Avoiding common errors

Try to write objects and protocol documentation **before** writing any other code; if you are having trouble documenting a predicate perhaps we need to go back to the design stage.

Try to avoid lengthy hierarchies. Besides performance penalties, composition is often a better choice over inheritance for defining new objects (Logtalk supports component-based programming through the use of categories). In addition, prototype-based hierarchies are conceptually simpler and more efficient than class-based hierarchies.

Dynamic predicates or dynamic entities are sometimes needed, but we should always try to minimize the use of non-logical features like destructive assignment (asserts and retracts).

Since each Logtalk entity is independently compiled, if an object inherits a dynamic or a meta-predicate predicate, then we must repeat the respective directives in order to ensure a correct compilation.

In general, Logtalk does not verify if a user predicate call/return arguments comply with the declared modes. On the other hand, Logtalk built-in predicates, built-in methods and message sending control structures are carefully checked for calling mode errors.

Logtalk error handling strongly depends on the ISO compliance of the chosen Prolog compiler. For instance, the error terms that are generated by some Logtalk built-in predicates assume that the Prolog built-ins behave as defined in the ISO standard regarding error conditions. In particular, if your Prolog compiler does not support a `read_term/3` built-in predicate compliant with the ISO Prolog Standard definition, then the current version of the Logtalk preprocessor will not be able to detect misspell variables in your source code.

Coding style guidlines

It is suggested that all code between an entity opening and closing directives be indented by one tab stop. When defining entity code, both directives and predicates, Prolog coding style guidelines may be applied. All Logtalk source files, examples, and standard library entities use four-space tabs for laying out code. Closed related entities should be defined in the same source file. Entities that might be useful in different contexts (such as library entities) are best defined in their own source files.

Logtalk scope

Logtalk, as an object-oriented extension to Prolog, shares with it the same preferred areas of application but also extends them with those areas where object-oriented features provide an advantage compared to plain Prolog. Among these areas we have:

Object-oriented programming teaching and researching
: Logtalk smooth learning curve, combined with support for both prototype and class-based programming, protocols, components via category-based composition, and other advanced object-oriented features allow a smooth introduction to object-oriented programming to people with a background in Prolog programming. The distribution of Logtalk source code using an open-source license provides a framework for people to learn and then modify to try out new ideas on object-oriented programming research.

Structured knowledge representations and knowledge-based systems
: Logtalk objects, coupled with event-driven programming features, enable easy implementation of frame-like systems and similar structured knowledge representations.

Blackboard systems, agent-based systems and systems with complex object relationships
: Logtalk support for event-driven programming can provide a basis for the dynamic and reactive nature of blackboard type applications.

Highly portable applications
: Logtalk is compatible with almost any modern Prolog compiler. Used as a way to provide Prolog with namespaces, it avoids the porting problems of most Prolog module systems. Platform, operating system, or compiler specific code can be isolated from the rest of the code by encapsulating it in objects with well defined interfaces.

Alternative to a Prolog module system
: Logtalk can be used as an alternative to a Prolog compiler module system. Any Prolog application that use modules can be converted to a Logtalk application, improving portability across Prolog compilers and taking advantage of the stronger reuse framework provided by Logtalk object-oriented features.

Integration with other programming languages
: Logtalk support for most key object-oriented features helps users integrating Prolog with object-oriented languages like C++, Java, or Smalltalk by providing an high-level mapping between the two languages.

XHTML + CSS Last updated on: March 18, 2006