Logtalk 3

User Manual

pmoura@logtalk.org
https://logtalk.org/

April 8, 2018

Updated for version 3.16.0

User Manual

Logtalk Features

Logtalk Nomenclature

Message Sending

Objects

Protocols

Categories

Predicates

Inheritance

Event-driven Programming

Multi-threading Programming

Error Handling

Documenting Logtalk Programs

Installing Logtalk

Writing, Running, and Debugging Logtalk Applications

Prolog Integration and Migration Guide

Logtalk main features

Some years ago, I decided that the best way to learn object-oriented programming was to build my own object-oriented language. Prolog always being my favorite language, I chose to extend it with object-oriented capabilities. Eventually this work has lead to the Logtalk system. The first public release of Logtalk 1.x occurred in February of 1995. Based on feedback by users and on the author subsequent work, the second major version went public in July of 1998.

Although this version of Logtalk shares many ideas and goals with previous 1.x versions, programs written for one version are not compatible with the other (however, conversion from previous versions can easily be accomplished in most cases). This is a consequence of the desire to have a more friendly system, with a very smooth learning curve, bringing Logtalk programming closer to traditional Prolog programming. There are, of course, also other important changes, that result in a more powerful and funnier system. Logtalk 2.x development provides the following features:

Integration of logic and object-oriented programming

Logtalk tries to bring together the main advantages of these two programming paradigms. On one hand, the object orientation allows us to work with the same set of entities in the successive phases of application development, giving us a way of organizing and encapsulating the knowledge of each entity within a given domain. On the other hand, logic programming allows us to represent, in a declarative way, the knowledge we have of each entity. Together, these two advantages allow us to minimize the distance between an application and its problem domain, turning the writing and maintenance of programming easier and more productive.

In a more pragmatically view, Logtalk objects provide Prolog with the possibility of defining several namespaces, instead of the traditional Prolog single database, addressing some of the needs of large software projects.

Integration of event-driven and object-oriented programming

Event-driven programming enables the building of reactive systems, where computing which takes place at each moment is a result of the observation of occurring events. This integration complements object-oriented programming, in which each computing is initiated by the explicit sending of a message to an object. The user dynamically defines what events are to be observed and establishes monitors for these events. This is specially useful when representing relationships between objects that imply constraints in the state of participating objects [Rumbaugh 87, Rumbaugh 88, Fornarino 89, Razek 92]. Other common uses are reflective applications like code debugging or profiling [Maes 87]. Predicates can be implicitly called when a spied event occurs, allowing programming solutions which minimize object coupling. In addition, events provide support for behavioral reflection and can be used to implement the concepts of *pointcut* and *advice* found on Aspect-Oriented Programming.

Support for component-based programming

Predicates can be encapsulated inside *categories* which can be imported by any object, without any code duplication and irrespective of object hierarchies. A category is a first-class encapsulation entity, at the same level as objects and protocols, which can be used as a component when building new objects. Thus, objects may be defined through composition of categories, which act as fine-grained units of code reuse. Categories may also extend existing objects. Categories can be used to implement *mixins* and *aspects*. Categories allows for code reuse between non-related objects, independent of hierarchy relations, in the same vein as protocols allow for interface reuse.

Support for both prototype and class-based systems

Almost any (if not all) object-oriented languages available today are either class-based or prototype-based [Lieberman 86], with a strong predominance of class-based languages. Logtalk provides support for both hierarchy types. That is, we can have both prototype and class hierarchies in the same application. Prototypes solve a problem of class-based systems where we sometimes have to define a class that will have only one instance in order to reuse a piece of code. Classes solves a dual problem in prototype based systems where it is not possible to encapsulate some code to be reused by other objects but not by the encapsulating object. Stand-alone objects, that is, objects that do not belong to any hierarchy, are a convenient solution to encapsulate code that will be reused by several unrelated objects.

Support for multiple object hierarchies

Languages like Smalltalk-80 [Goldberg 83], Objective-C [Cox 86] and Java [Joy et al. 00] define a single hierarchy rooted in a class usually named `Object`. This makes it easy to ensure that all objects share a common behavior but also tends to result in lengthy hierarchies where it is difficult to express objects which represent exceptions to default behavior. In Logtalk we can have multiple, independent, object hierarchies. Some of them can be prototype-based while others can be class-based. Furthermore, stand-alone objects provide a simple way to encapsulate utility predicates that do not need or fit in an object hierarchy.

Separation between interface and implementation

This is an expected (should we say standard ?) feature of almost any modern programming language. Logtalk provides support for separating interface from implementation in a flexible way: protocol directives can be contained in an object, a category or a protocol (first-order entities in Logtalk) or can be spread in both objects, categories and protocols.

Private, protected and public inheritance

Logtalk supports private, protected and public inheritance in a similar way to C++ [Stroustrup 86], enabling us to restrict the scope of inherited, imported or implemented predicates (by default inheritance is public).

Private, protected and public object predicates

Logtalk supports data hiding by implementing private, protected and public object predicates in a way similar to C++ [Stroustrup 86]. Private predicates can only be called from the container object. Protected predicates can be called by the container object or by the container descendants. Public predicates can be called from any object.

Parametric objects

Object names can be compound terms (instead of atoms), providing a way to parameterize object predicates. Parametric objects are implemented in a similar way to `L&O` [McCabe 92], `OL(P)` [Fromherz 93] or `SICStus Objects` [SICStus 95] (however, access to parameter values is done via a built-in method instead of making the parameters scope global over the whole object). Parametric objects allows us to treat any predicate clause as defining an *instantiation* of a parametric object. Thus, a parametric object allows us to encapsulate and associate any number of predicates with a compound term.

High level multi-threading programming support

High level multi-threading programming is available when running Logtalk with selected back-end Prolog compilers, allowing objects to support both synchronous and asynchronous messages. Logtalk allows programmers to take advantage of modern multi-processor and multi-core computers without bothering with the details of creating and destroying threads, implement thread communication, or synchronizing threads.

Smooth learning curve

Logtalk has a smooth learning curve, by adopting standard Prolog syntax and by enabling an incremental learning and use of most of its features.

Compatibility with most Prologs and the ISO standard

The Logtalk system has been designed to be compatible with most Prolog compilers and, in particular, with the ISO Prolog standard [ISO 95]. It runs in almost any computer system with a modern Prolog compiler.

Performance

The current Logtalk implementation works as a pre-processor: Logtalk source files are first compiled to Prolog source files, which are then compiled by the chosen Prolog compiler. Therefore, Logtalk performance necessarily depends on the back-end Prolog compiler. The Logtalk compiler respects the programmers choices when writing efficient code that takes advantage of tail recursion and first-argument indexing.

As an object-oriented language, Logtalk uses both static binding and dynamic binding for matching messages and methods. Furthermore, Logtalk entities (objects, protocols, and categories) are independently compiled, allowing for a very flexible programming development. Entities can be edited, compiled, and loaded at runtime, without necessarily implying recompilation of all related entities.

When dynamic binding is used, the Logtalk runtime engine implements caching of method lookups (including messages to *self* and *super* calls), ensuring a performance level close to what could be achieved when using static binding.

Logtalk scope

Logtalk, being a superset of 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:

Logic and object-oriented programming teaching and researching
: Logtalk smooth learning curve, combined with support for both prototype and class-based programming, protocols, components or aspects 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. In addition, the Logtalk distribution includes plenty of programming examples that can be used in the classroom for teaching logic and object-oriented programming concepts.

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 encapsulation and 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.

Nomenclature

Depending on your Object-oriented Programming background (or lack of it), you may find Logtalk nomenclature either familiar or at odds with the terms used in other languages. In addition, being a superset of Prolog, terms such as *predicate* and *method* are often used interchangeably. Logtalk inherits most of its nomenclature from Smalltalk, arguably (and somehow sadly) not the most popular OOP language nowadays. In this section, we map nomenclatures from popular OOP languages such as C++ and Java to the Logtalk nomenclature.

C++ nomenclature

There are several C++ glossaries available on the Internet. The list that follows relates the most commonly used C++ terms with their Logtalk equivalents.

abstract class
: Logtalk uses an *operational* definition of abstract class: any class that does not inherit a method for creating new instances is an abstract class. Moreover, Logtalk supports *interfaces*/*protocols*, which are often a better way to provide the functionality of C++ abstract classes.

base class
: Logtalk uses the term *superclass* with the same meaning.

data member
: Logtalk uses *predicates* for representing both behavior and data.

constructor function
: There are no special methods for creating new objects in Logtalk. Instead, Logtalk provides a built-in predicate, `create_object/4`, which is often used to define more sophisticated object creation predicates.

derived class
: Logtalk uses the term *subclass* with the same meaning.

destructor function
: There are no special methods for deleting new objects in Logtalk. Instead, Logtalk provides a built-in predicate, `abolish_object/1`, which is often used to define more sophisticated object deletion predicates.

friend function
: Not supported in Logtalk. Nevertheless, see the manual section on *meta-predicates*.

instance
: In Logtalk, an instance can be either created dynamically at runtime or defined statically in a source file in the same way as classes.

member
: Logtalk uses the term predicate.

member function
: Logtalk uses predicates for representing both behavior and data.

namespace
: Logtalk does not support multiple identifier namespaces. All Logtalk entity identifiers share the same namespace (Logtalk entities are objects, categories, and protocols).

nested class
: Logtalk does not support nested classes.

template
: Logtalk supports *parametric objects*, which allows you to get the similar functionality of templates at runtime.

this
: Logtalk uses the built-in context method `self/1` for retrieving the current instance. Logtalk also provides a `this/1` method but for returning the class containing the method being executed. Why the name clashes? Well, the notion of *self* was inherited from Smalltalk, which predates C++.

virtual member function
: There is no `virtual` keyword in Logtalk. By default, Logtalk uses dynamic binding for locating both method declarations and method definitions. Moreover, methods that are declared but not defined simply fail when called.

Java nomenclature

There are several Java glossaries available on the Internet. The list that follows relates the most commonly used Java terms with their Logtalk equivalents.

abstract class
: Logtalk uses an *operational* definition of abstract class: any class that does not inherit a method for creating new instances is an abstract class. I.e. there is no `abstract` keyword in Logtalk.

abstract method
: In Logtalk, you may simply declare a method (predicate) in a class without defining it, leaving its definition to some descendant sub-class.

assertion
: There is no `assertion` keyword in Logtalk. Assertions are supported using Logtalk compilation hooks and developer tools.

extends
: There is no `extends` keyword in Logtalk. Class inheritance is indicated using *specialization* relations. Moreover, the *extends* relation is used in Logtalk to indicate protocol, category, or prototype extension.

interface
: Logtalk uses the term *protocol* with the same meaning.

callback method
: Logtalk supports *event-driven programming*, the most common use context of callback methods.

class method
: Class methods may be implemented in Logtalk by using a metaclass for the class and defining the class methods in the metaclass. I.e. class methods are simply instance methods of the class metaclass.

class variable
: True class variables may be implemented in Logtalk by using a metaclass for the class and defining the class variables in the class. I.e. class variables are simply instance variables of the class metaclass. Shared instance variables may be implemented by using the built-in database methods (which can be used to implement variable

assignment) to access and updated a single occurrence of the variable stored in the class (there is no `static` keyword in Logtalk).

constructor
There are no special methods for creating new objects in Logtalk. Instead, Logtalk provides a built-in predicate, `create_object/4`, which is often used to define more sophisticated object creation predicates.

final
There is no `final` keyword in Logtalk; methods may always be redefined in subclasses (and instances!).

inner class
Inner classes are not supported in Logtalk.

instance
In Logtalk, an instance can be either created dynamically at runtime or defined statically in a source file in the same way as classes.

method
Logtalk uses the term *predicate* interchangeably with the term *method*.

method call
Logtalk usually uses the expression *message sending* for method calls, true to its Smalltalk heritage.

method signature
Logtalk selects the method/predicate to execute in order to answer a method call based only on the method name (functor) and number of arguments (arity). Logtalk (and Prolog) are not typed languages in the same sense as Java.

reflection
Logtalk supports both *structural reflection* (using a set of built-in predicates and built-in methods) and *behavioral reflection* (using event-driven programming).

static
There is no `static` keyword in Logtalk. See the entries on *class methods* and *class variables*.

super
Instead of a `super` keyword, Logtalk provides a *super* operator, `^^/1`, for calling overridden methods.

synchronized
Logtalk supports *multi-threading programming* in selected Prolog compilers, including a `synchronized/1` predicate directive. Logtalk allows you to synchronize a predicate or a set of predicates using per-predicate or per-predicate-set *mutexes*.

this
Logtalk uses the built-in context method `self/1` for retrieving the current instance. Logtalk also provides a `this/1` method but for returning the class containing the method being executed. Why the name clashes? Well, the notion of *self* was inherited from Smalltalk, which predates Java.

Message sending

Messages allows us to call object predicates. Logtalk uses the same nomenclature found in other object-oriented programming languages such as Smalltalk. Therefore, the terms *predicate* and *method* are often used interchangeably when referring to predicates defined inside objects and categories. A message must always match a predicate within the scope of the sender object.

Note that message sending is only the same as calling an object's predicate if the object does not inherit (or import) predicate definitions from other objects (or categories). Otherwise, the predicate definition that will be executed may depend on the relationships of the object with other Logtalk entities.

Operators used in message sending

Logtalk declares the following operators for the message sending control constructs:

```
:- op(600, xfy, ::).
:- op(600,  fy, ::).
:- op(600,  fy, ^^).
```

It is assumed that these operators remain active (once the Logtalk compiler and runtime files are loaded) until the end of the Prolog session (this is the usual behavior of most Prolog compilers). Note that these operator definitions are compatible with the pre-defined operators in the Prolog ISO standard.

Sending a message to an object

Sending a message to an object is accomplished by using the `::/2` control construct:

```
| ?- Object::Message.
```

The message must match a public predicate declared for the receiving object. The message may also correspond to a protected or private predicate if the *sender* matches the predicate scope container. If the predicate is declared but not defined, the message simply fails (as per the closed-world assumption).

Delegating a message to an object

It is also possible to send a message to an object while preserving the original *sender* by using the `[]/1` delegation control construct:

```
..., [Object::Message], ....
```

This control construct can only be used within objects and categories (at the interpreter top-level, the *sender* is always the pseudo-object *user* so using this control construct would be equivalent to use the `::/2` message sending control construct).

Sending a message to *self*

While defining a predicate, we sometimes need to send a message to *self*, i.e., to the same object that has received the original message. This is done in Logtalk through the `::/1` control construct:

```
::Message
```

The message must match either a public or protected predicate declared for the receiving object or a private predicate within the scope of the *sender* otherwise an error will be thrown (see the Reference Manual for details). If the message is sent from inside a category or if we are using private inheritance, then the message may also match a private predicate. Again, if the predicate is declared but not defined, the message simply fails (as per the closed-world assumption).

Broadcasting

In the Logtalk context, broadcasting is interpreted as the sending of several messages to the same object. This can be achieved by using the message sending method described above. However, for convenience, Logtalk implements an extended syntax for message sending that may improve program readability in some cases. This extended syntax uses the `(,)/2`, `(;)/2`, and `(->)/2` control constructs. For example, if we wish to send several messages to the same object, we can write:

```
| ?- Object::(Message1, Message2, ...).
```

This is semantically equivalent to:

```
| ?- Object::Message1, Object::Message2, ... .
```

This extended syntax may also be used with the `::/1` message sending control construct.

Calling imported and inherited predicate definitions

When redefining a predicate, sometimes we need to call the inherited definition in the new code. This functionality, introduced by the Smalltalk language through the `super` primitive, is available in Logtalk using the `^^/1` control construct:

```
^^Predicate
```

Most of the time we will use this control construct by instantiating the pattern:

```
Predicate :-
    ...,            % do something
    ^^Predicate,    % call inherited definition
    ... .           % do something more
```

This control construct is generalized in Logtalk where it may be used to call any imported or inherited predicate definition. This control construct may be used within objects and categories. When combined with static binding, this control construct

allows imported and inherited predicates to be called with the same performance of local predicates. As with the message sending control constructs, the `^^/1` call simply fails when the predicate is declared but not defined (as per the closed-world assumption).

Message sending and event generation

Every message sent using the `::/2` control construct generates two events, one before and one after the message execution. Messages that are sent using the `::/1` (message to *self*) control construct or the `^^/1` super mechanism described above do not generate any events. The rational behind this distinction is that messages to *self* and *super* calls are only used internally in the definition of methods or to execute additional messages with the same target object (represented by *self*). In other words, events are only generated when using an object's public interface; they cannot be used to break object encapsulation.

If we need to generate events for a public message sent to *self*, then we just need to write something like:

```
Predicate :-
    ...,
    self(Self),       % get self reference
    Self::Message,    % send a message to self using ::/2
    ... .
```

If we also need the sender of the message to be other than the object containing the predicate definition, we can write:

```
Predicate :-
    ...,
    self(Self),       % send a message to self using ::/2
    {Self::Message},  % sender will be the pseudo-object user
    ... .
```

When events are not used, is possible to turn off event generation on a per object basis by using the `events/1` compiler flag. See the section on event-driven programming for more details.

Message sending performance

Logtalk supports both static binding and dynamic binding. Static binding is used whenever messages are sent (using `::/2`) to static objects already loaded and with the `optimize` compiler flag turned on. When that is not the case (or when using `::/1`), Logtalk uses dynamic binding coupled with a caching mechanism that avoids repeated lookups of predicate declarations and predicate definitions. This is a solution common to other programming languages supporting dynamic binding. Message lookups are automatically cached the first time a message is sent. Cache entries are automatically removed when loading entities or using Logtalk dynamic features that invalidate the cached lookups.

Whenever static binding is used, message sending performance is roughly the same as a predicate call in plain Prolog. When discussing Logtalk dynamic binding performance, two distinct cases should be considered: messages sent by the user from the top-level interpreter and messages sent from compiled objects. In addition, the message declaration and definition lookups may, or may not be already cached by the runtime engine. In what follows, we will assume that the message lookups are already cached.

Translating message processing to predicate calls

In order to better understand the performance tradeoffs of using Logtalk dynamic binding when compared to plain Prolog or to Prolog module systems, is useful to translate message processing in terms of predicate calls. However, in doing this, we should keep in mind that the number of predicate calls is not necessarily proportional to the time taken to execute them.

With event-support turned on, a message sent from a compiled object to another object translates to three predicate calls:

checking for *before* events
: one call to the built-in predicate `\+/1`, assuming that no events are defined

method call using the cached lookup
: one call to a dynamic predicate (the cache entry)

checking for *after* events
: one call to the built-in predicate `\+/1`, assuming that no events are defined

Given that events can be dynamically defined at runtime, there is no room for reducing the number of predicate calls without turning off support for event-driven programming. When events are defined, the number of predicate calls grows proportional to the number of events and event handlers (monitors). Event-driven programming support can be switched off for specific object using the compiler flag `events/1`. Doing so, reduces the number of predicate calls from three to just one.

Messages to *self* and *super* calls are transparent regarding events and, as such, imply only one predicate call (to the cache entry, a dynamic predicate).

When a message is sent by the user from the top-level interpreter, Logtalk needs to perform a runtime translation of the message term in order to prove the corresponding goal. Thus, while sending a message from a compiled object corresponds to either three predicate calls (event-support on) or one predicate call (event-support off), the same message sent by the user from the top-level interpreter necessarily implies an overhead. Considering the time taken for the user to type the goal and read the reply, this overhead is of no practical consequence.

When a message is not cached, the number of predicate calls depends on the number of steps needed for the Logtalk runtime engine to lookup the corresponding predicate scope declaration (to check if the message is valid) and then to lookup a predicate definition for answering the message.

Processing time

Not all predicate calls take the same time. Moreover, the time taken to process a specific predicate call depends on the Prolog compiler implementation details. As such, the only valid performance measure is the time taken for processing a message.

The usual way of measuring the time taken by a predicate call is to repeat the call a number of times and than to calculate the average time. A sufficient large number of repetitions would hopefully lead to an accurate measure. Care should be taken to subtract the time taken by the repetition code itself. In addition, we should be aware of any limitations of the predicates used to measure execution times. One way to make sense of numbers we get is to repeat the test with the same predicate using plain Prolog and with the predicate encapsulated in a module.

A simple predicate for helping benchmarking predicate calls could be:

```
benchmark(N, Goal) :-
    repeat(N),
        call(Goal),
    fail.

benchmark(_, _).
```

The rational of using a failure-driven loop is to try to avoid any interference on our timing measurements from garbage-collection or memory expansion mechanisms. Based on the predicate `benchmark/2`, we may define a more convenient predicate for performing our benchmarks. For example:

```
benchmark(Goal) :-
    N = 10000000,                   % some sufficiently large number of repetitions
    write('Number of repetitions: '), write(N), nl,
    get_cpu_time(Seconds1),         % replace by your Prolog-specific predicate
    benchmark(N, Goal),
    get_cpu_time(Seconds2),
    Average is (Seconds2 - Seconds1)/N,
    write('Average time per call: '), write(Average), write(' seconds'), nl,
    Speed is 1.0/Average,
    write('Number of calls per second: '), write(Speed), nl.
```

We can get a baseline for our timings by doing:

```
| ?- benchmark(true).
```

For comparing message sending performance across several Prolog compilers, we would call the `benchmark/1` predicate with a suitable argument. For example:

```
| ?- benchmark(list::length([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], _)).
```

For comparing message sending performance with predicate calls in plain Prolog and with calls to predicates encapsulated in modules, we should use exactly the same predicate definition in the three cases.

It should be stressed that message sending is only one of the factors affecting the performance of a Logtalk application (and often not the most important one). The strengths and limitations of the chosen Prolog compiler play a crucial role on all aspects of the development, reliability, usability, and performance of a Logtalk application. It is advisable to take advantage of the Logtalk wide compatibility with most Prolog compilers to test for the best match for developing your Logtalk applications.

Objects

The main goal of Logtalk objects is the encapsulation and reuse of predicates. Instead of a single database containing all your code, Logtalk objects provide separated namespaces or databases allowing the partitioning of code in more manageable parts. Logtalk does not aim to bring some sort of new dynamic state change concept to Logic Programming or Prolog.

In Logtalk, the only pre-defined objects are the built-in objects `user` and `logtalk`, which are described at the end of this section.

Objects, prototypes, classes, and instances

There are only three kinds of encapsulation entities in Logtalk: objects, protocols, and categories. Logtalk uses the term *object* in a broad sense. The terms *prototype*, *parent*, *class*, *subclass*, *superclass*, *metaclass*, and *instance* always designate an object. Different names are used to emphasize the *role* played by an object in a particular context. I.e. we use a term other than object when we want to make the relationship with other objects explicit. For example, an object with an *instantiation* relation with other object plays the role of an *instance*, while the instantiated object plays the role of a *class*; an object with a *specialization* relation with other object plays the role of a *subclass*, while the specialized object plays the role of a *superclass*; an object with an *extension* relation with other object plays the role of a *prototype*, the same for the extended object. A *stand-alone* object, i.e. an object with no relations with other objects, is always interpreted as a prototype. In Logtalk, entity relations essentially define *patterns* of code reuse. An entity is compiled accordingly to the roles it plays.

Logtalk allows you to work from standalone objects to any kind of hierarchy, either class-based or prototype-based. You may use single or multiple inheritance, use or forgo metaclasses, implement reflective designs, use parametric objects, and take advantage of protocols and categories (think components).

Prototypes

Prototypes are either self-defined objects or objects defined as extensions to other prototypes with whom they share common properties. Prototypes are ideal for representing one-of-a-kind objects. Prototypes usually represent concrete objects in the application domain. When linking prototypes using *extension* relations, Logtalk uses the term *prototype hierarchies* although most authors prefer to use the term *hierarchy* only with class generalization/specialization relations. In the context of logic programming, prototypes are often the ideal replacement for modules.

Classes

Classes are used to represent abstractions of common properties of sets of objects. Classes provide an ideal structuring solution when you want to express hierarchies of abstractions or work with many similar objects. Classes are used indirectly through *instantiation*. Contrary to most object-oriented programming languages, instances can be created both dynamically at runtime or defined in a source file like other objects.

Defining a new object

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 object. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the adapter files. Intermediate Prolog source files (generated by the Logtalk compiler) have, by default, a `_lgt` suffix and a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding adapter file. For instance, we may define an object named `vehicle` and save it in a `vehicle.lgt` source file which will be compiled to a `vehicle_lgt.pl` Prolog file (depending on the backend compiler, the names of the intermediate Prolog files may include a directory hash).

Object names can be atoms or compound terms (when defining parametric objects, see below). Objects, categories, and protocols share the same name space: we cannot have an object with the same name as a protocol or a category.

Object code (directives and predicates) is textually encapsulated by using two Logtalk directives: `object/1-5` and `end_object/0`. The most simple object will be one that is self-contained, not depending on any other Logtalk entity:

```
:- object(Object).
    ...
:- end_object.
```

If an object implements one or more protocols then the opening directive will be:

```
:- object(Object,
    implements([Protocol1, Protocol2, ...])).
    ...
:- end_object.
```

An object can import one or more categories:

```
:- object(Object,
    imports([Category1, Category2, ...])).
    ...
:- end_object.
```

If an object both implements protocols and imports categories then we will write:

```
:- object(Object,
    implements([Protocol1, Protocol2, ...]),
    imports([Category1, Category2, ...])).
    ...
:- end_object.
```

In object-oriented programming objects are usually organized in hierarchies that enable interface and code sharing by inheritance. In Logtalk, we can construct prototype-based hierarchies by writing:

```
:- object(Prototype,
    extends(Parent)).
    ...
:- end_object.
```

We can also have class-based hierarchies by defining instantiation and specialization relations between objects. To define an object as a class instance we will write:

```
:- object(Object,
    instantiates(Class)).
    ...
:- end_object.
```

A class may specialize another class, its superclass:

```
:- object(Class,
    specializes(Superclass)).
    ...
:- end_object.
```

If we are defining a reflexive system where every class is also an instance, we will probably be using the following pattern:

```
:- object(Class,
    instantiates(Metaclass),
    specializes(Superclass)).
    ...
:- end_object.
```

In short, an object can be a *stand-alone* object or be part of an object hierarchy. The hierarchy can be prototype-based (defined by extending other objects) or class-based (with instantiation and specialization relations). An object may also implement one or more protocols or import one or more categories.

A *stand-alone* object (i.e. an object with no extension, instantiation, or specialization relations with other objects) is always compiled as a prototype, that is, a self-describing object. If we want to use classes and instances, then we will need to

specify at least one instantiation or specialization relation. The best way to do this is to define a set of objects that provide the basis of a reflective system [Cointe 87, Moura 94]. For example:

```
:- object(object,               % default root of the inheritance graph
    instantiates(class)).       % predicates common to all objects
    ...
:- end_object.

:- object(class,                % default metaclass for all classes
    instantiates(class),        % predicates common to all instantiable classes
    specializes(abstract_class)).
    ...
:- end_object.

:- object(abstract_class,       % default metaclass for all abstract classes
    instantiates(class),        % predicates common to all classes
    specializes(object)).
    ...
:- end_object.
```

Note that with these instantiation and specialization relations, `object`, `class`, and `abstract_class` are, at the same time, classes and instances of some class. In addition, each object inherits its own predicates and the predicates of the other two objects without any inheritance loop problems.

When a full-blown reflective system solution is not needed, the above scheme can be simplified by making an object an instance of itself, i.e. by making a class its own metaclass. For example:

```
:- object(class,
    instantiates(class)).
    ...
:- end_object.
```

We can use, in the same application, both prototype and class-based hierarchies (and freely exchange messages between all objects). We cannot however mix the two types of hierarchies by, e.g., specializing an object that extends another object in this current Logtalk version.

Parametric objects

Parametric objects have a compound term for name instead of an atom. This compound term usually contains free variables that can be instantiated when sending or as a consequence of sending a message to the object, thus acting as object parameters. The object predicates can then be coded to depend on those parameters, which are logical variables shared by all object predicates. When an object state is set at object creation and never changed, parameters provide a better solution than using the object's database via asserts. Parametric objects can also be used to associate a set of predicates to terms that share a common functor and arity.

In order to give access to an object parameter, Logtalk provides the `parameter/2` built-in local method:

```
:- object(Functor(Arg1, Arg2, ...)).

    ...

    Predicate :-
        ...,
        parameter(Number, Value),
        ... .
```

An alternative solution is to use the built-in local method `this/1`. For example:

```
:- object(foo(Arg)).

    ...

    bar :-
        ...,
        this(foo(Arg)),
        ... .
```

Both solutions are equally efficient as calls to the methods `this/1` and `parameter/2` are usually compiled inline into a clause head unification. The drawback of this second solution is that we must check all calls of `this/1` if we change the object name. Note that we can't use these method with the message sending operators (`::/2`, `::/1`, or `^^/1`).

A third alternative to access object parameters is to use *parameter variables*. Although parameter variables introduce a concept of entity global variables, they allow object parameters to be added, rearranged, or removed without requiring any changes to the clauses that refer to them. Note that using parameter variables doesn't change the fact that entity parameters are logical variables.

When storing a parametric object in its own source file, the convention is to name the file after the object, with the object arity appended. For instance, when defining an object named `sort(Type)`, we may save it in a `sort_1.lgt` text file. This way it is easy to avoid file name clashes when saving Logtalk entities that have the same functor but different arity.

Compound terms with the same functor and with the same number of arguments as a parametric object identifier may act as *proxies* to a parametric object. Proxies may be stored on the database as Prolog facts and be used to represent different instantiations of a parametric object identifier. Logtalk provides a convenient notation for accessing proxies represented as Prolog facts when sending a message:

```
{Proxy}::Message
```

In this context, the proxy argument is proved as a plain Prolog goal. If successful, the message is sent to the corresponding parametric object. Typically, the proof allows retrieving of parameter instantiations. This construct can either be used with a proxy argument that is sufficiently instantiated in order to unify with a single Prolog fact or with a proxy argument that unifies with several facts on backtracking.

Finding defined objects

We can find, by backtracking, all defined objects by calling the `current_object/1` built-in predicate with a non-instantiated variable:

```
| ?- current_object(Object).
```

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

Creating a new object in runtime

An object can be dynamically created at runtime by using the `create_object/4` built-in predicate:

```
| ?- create_object(Object, Relations, Directives, Clauses).
```

The first argument should be either a variable or the name of the new object (a Prolog atom or compound term, which must not match any existing entity name). The remaining three arguments correspond to the relations described in the opening object directive and to the object code contents (directives and clauses).

For instance, the call:

```
| ?- create_object(foo, [extends(bar)], [public(foo/1)], [foo(1), foo(2)]).
```

is equivalent to compiling and loading the object:

```
:- object(foo,
    extends(bar)).

    :- dynamic.

    :- public(foo/1).

    foo(1).
    foo(2).

:- end_object.
```

If we need to create a lot of (dynamic) objects at runtime, then is best to define a metaclass or a prototype with a predicate that will call this built-in predicate to make new objects. This predicate may provide automatic object name generation, name checking, and accept object initialization options.

Abolishing an existing object

Dynamic objects can be abolished using the `abolish_object/1` built-in predicate:

```
| ?- abolish_object(Object).
```

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

Object directives

Object directives are used to set initialization goals, define object properties, to document an object dependencies on other Logtalk entities, and to load the contents of files into an object.

Object initialization

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

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

The argument can be any valid Prolog or Logtalk goal, including a message to other object. For example:

```
:- object(foo).

    :- initialization(init).
    :- private(init/0).

    init :-
        ... .

    ...

:- end_object.
```

Or:

```
:- object(assembler).

    :- initialization(control::start).
    ...

:- end_object.
```

The initialization goal can also be a message to *self* in order to call an inherited or imported predicate. For example, assuming that we have a `monitor` category defining a `reset/0` predicate:

```
:- object(profiler,
    imports(monitor)).

    :- initialization(::reset).
    ...

:- end_object.
```

Note, however, that descendant objects do not inherit initialization directives. In this context, *self* denotes the object that contains the directive. Also note that by initialization we do not necessarily mean setting an object dynamic state.

Dynamic objects

Similar to Prolog predicates, an object can be either static or dynamic. An object created during the execution of a program is always dynamic. An object defined in a file can be either dynamic or static. Dynamic objects are declared by using the `dynamic/0` directive in the object source code:

```
:- dynamic.
```

The directive must precede any predicate directives or clauses. Please be aware that using dynamic code results in a performance hit when compared to static code. We should only use dynamic objects when these need to be abolished during program execution. In addition, note that we can declare and define dynamic predicates within a static object.

Object documentation

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

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

See the documenting Logtalk programs section for details.

Loading files into an object

The `include/1` directive can be used to load the contents of a file into an object. A typical usage scenario is to load a plain Prolog database into an object thus providing a simple way to encapsulate it. For example, assume a `cities.pl` file defining facts for a `city/4` predicate. We could define a wrapper for this database by writing:

```
:- object(cities).

 :- public(city/4).

 :- include(dbs('cities.pl')).

:- end_object.
```

The `include/1` directive can also be used when creating an object dynamically. For example:

```
| ?- create_object(cities, [], [public(city/4), include(dbs('cities.pl'))], []).
```

Object relationships

Logtalk provides six sets of built-in predicates that enable us to query the system about the possible relationships that an object may have with other entities.

The built-in predicates `instantiates_class/2` and `instantiates_class/3` can be used to query all instantiation relations:

```
| ?- instantiates_class(Instance, Class).
```

or, if we want to know the instantiation scope:

```
| ?- instantiates_class(Instance, Class, Scope).
```

Specialization relations can be found by using either the `specializes_class/2` or the `specializes_class/3` built-in predicates:

```
| ?- specializes_class(Class, Superclass).
```

or, if we want to know the specialization scope:

```
| ?- specializes_class(Class, Superclass, Scope).
```

For prototypes, we can query extension relations with the `extends_object/2` or the `extends_object/3` built-in predicates:

```
| ?- extends_object(Object, Parent).
```

or, if we want to know the extension scope:

```
| ?- extends_object(Object, Parent, Scope).
```

In order to find which objects import which categories we can use the built-in predicates `imports_category/2` or `imports_category/3`:

```
| ?- imports_category(Object, Category).
```

or, if we want to know the importation scope:

```
| ?- imports_category(Object, Category, Scope).
```

To find which objects implements which protocols we can use the `implements_protocol/2-3` and `conforms_to_protocol/2-3` built-in predicates:

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

or, if we also want inherited protocols:

```
| ?- conforms_to_protocol(Object, Protocol, Scope).
```

Note that, if we use a non-instantiated variable for the first argument, we will need to use the `current_object/1` built-in predicate to ensure that the entity returned is an object and not a category.

To find which objects are explicitly complemented by categories we can use the `complements_object/2` built-in predicate:

```
| ?- complements_object(Category, Object).
```

Note that more than one category may explicitly complement a single object.

Object properties

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

```
| ?- object_property(Object, Property).
```

The following object properties are supported:

`static`
: The object is static

`dynamic`
: The object is dynamic (and thus can be abolished in runtime by calling the `abolish_object/1` built-in predicate)

`built_in`
: The object is a built-in object (and thus always available)

`threaded`
: The object supports/makes multi-threading calls

`file(Path)`
: Absolute path of the source file defining the object (if applicable)

`file(Basename, Directory)`
: Basename and directory of the source file defining the object (if applicable)

`lines(BeginLine, EndLine)`
: Source file begin and end lines of the object definition (if applicable)

`context_switching_calls`
: The object supports context switching calls (i.e. can be used with the `<</2` debugging control construct)

`dynamic_declarations`
: The object supports dynamic declarations of predicates

`events`
: Messages sent from the object generate events

`source_data`
: Source data available for the object

`complements(Permission)`
: The object supports complementing categories with the specified permission (`allow` or `restrict`)

`complements`
: The object supports complementing categories

`public(Predicates)`
: List of public predicates declared by the object

`protected(Predicates)`
: List of protected predicates declared by the object

`private(Predicates)`
: List of private predicates declared by the object

`declares(Predicate, Properties)`
: List of properties for a predicate declared by the object

`defines(Predicate, Properties)`
: List of properties for a predicate defined by the object

`includes(Predicate, Entity, Properties)`
: List of properties for an object multifile predicate that are defined in the specified entity (the properties include `number_of_clauses(Number)`, `number_of_rules(Number)`, and `line_count(Line)` with `Line` being the begin line of the multifile predicate clause)

`provides(Predicate, Entity, Properties)`
: List of properties for other entity multifile predicate that are defined in the object (the properties include `number_of_clauses(Number)`, `number_of_rules(Number)`, and `line_count(Line)` with `Line` being the begin line of the multifile predicate clause)

`alias(Predicate, Properties)`
: List of properties for a predicate alias declared by the object (the properties include `for(Original)`, `from(Entity)`, `non_terminal(NonTerminal)`, and `line_count(Line)` with `Line` being the begin line of the alias directive)

`calls(Call, Properties)`
: List of properties for predicate calls made by the object (`Call` is either a predicate indicator or a control construct such as `::/1-2` or `^^/1` with a predicate indicator as argument; note that `Call` may not be ground in case of a call to a control construct where its argument is only know at runtime; the properties include `caller(Caller)`, `alias(Alias)`, and `line_count(Line)` with both `Caller` and `Alias` being predicate indicators and `Line` being the begin line of the predicate clause or directive making the call)

`updates(Predicate, Properties)`
: List of properties for dynamic predicate updates (and also access using the `clause/2` predicate) made by the object (`Predicate` is either a predicate indicator or a control construct such as `::/1-2` or `:/2` with a predicate indicator as argument; note that `Predicate` may not be ground in case of a control construct argument only know at runtime; the properties include `updater(Updater)`, `alias(Alias)`, and `line_count(Line)` with `Updater` being a (possibly multifile) predicate indicator, `Alias` being a predicate indicator, and `Line` being the begin line of the predicate clause or directive updating the predicate)

`number_of_clauses(Number)`
: Total number of predicate clauses defined in the object at compilation time (includes both user-defined clauses and auxiliary clauses generated by the compiler or by the expansion hooks)

`number_of_rules(Number)`
: Total number of predicate rules defined in the object at compilation time (includes both user-defined rules and auxiliary rules generated by the compiler or by the expansion hooks)

`number_of_user_clauses(Number)`
: Total number of user-defined predicate clauses defined in the object at compilation time

`number_of_user_rules(Number)`
: Total number of user-defined predicate rules defined in the object at compilation time

When a predicate is called from an `initialization/1` directive, the argument of the `caller/1` property is `:-/1`.

Some of the properties such as line numbers are only available when the object is defined in a source file compiled with the `source_data` flag turned on.

The properties that return the number of clauses (rules) report the clauses (rules) *textually defined in the object* for both multifile and non-multifile predicates. Thus, these numbers exclude clauses (rules) for multifile predicates *contributed* by other entities.

Built-in objects

Logtalk defines some built-in objects that are always available for any application.

The built-in pseudo-object *user*

Logtalk defines a built-in, pseudo-object named `user` that virtually contains all user predicate definitions not encapsulated in a Logtalk entity. These predicates are assumed to be implicitly declared public. Messages sent from this pseudo-object, which includes messages sent from the top-level interpreter, always generate events. Defining complementing categories for this pseudo-object is not supported.

With some of the backend Prolog compilers that support a module system, it is possible to load (the) Logtalk (compiler/runtime) into a module other than the pseudo-module *user*. In this case, the Logtalk pseudo-object *user* virtually contains all user predicate definitions defined in the module where Logtalk was loaded.

The built-in object *logtalk*

Logtalk defines a built-in object named `logtalk` that provides structured message printing mechanism predicates, structured question asking predicates, debugging event predicates, predicates for accessing the internal database of loaded files and their properties, and also a set of low-level utility predicates normally used when defining hook objects.

The following predicates are defined:

`expand_library_path(Library, Path)`
: Expands a file specification in library notation to a full operating-system path.

`loaded_file(Path)`
: Returns the full path of a currently loaded source file.

`loaded_file_property(Path, Property)`
: Returns a property for a currently loaded source file. Valid properties are `basename/1`, `directory/1`, `flags/1` (explicit flags used when the file was loaded), `text_properties/1` (list, possibly empty, whose possible elements are `encoding/1` and `bom/1`), `target/1` (full path for the Prolog file generated by the compilation of the loaded source file), `modified/1` (time stamp that should be treated as an opaque term but that may be used for comparisons), `parent/1` (parent file, if it exists, that loaded the file; a file may have multiple parents), and `library/1` (library name when there is a library whose location is the same as the loaded file directory).

`compile_aux_clauses(Clauses)`
: Compiles a list of clauses in the context of the entity under compilation. This method is usually called from `goal_expansion/2` hooks in order to compile auxiliary clauses generated for supporting an expanded goal. The compilation of the clauses avoids the risk of making the predicate whose clause is being goal-expanded discontiguous by accident.

`entity_prefix(Entity, Prefix)`
: Converts an entity identifier into its internal prefix or an internal prefix into an entity identifier.

`compile_predicate_heads(Heads, Entity, TranslatedHeads, ContextArgument)`
: Compiles a predicate head or a list of predicate heads in the context of the specified entity or in the context of the entity being compiled when `Entity` is not instantiated.

`compile_predicate_indicators(PredicateIndicators, Entity, TranslatedPredicateIndicators)`
: Compiles a predicate indicator or a list of predicate indicators in the context of the specified entity or in the context of the entity being compiled when `Entity` is not instantiated.

`decompile_predicate_heads(TranslatedHeads, Entity, EntityType, Heads)`
: Decompiles a compiled predicate head or a list of compiled predicate heads returning the entity, entity type, and source level heads. Requires the entity to be currently loaded.

`decompile_predicate_indicators(TranslatedPredicateIndicators, Entity, EntityType, PredicateIndicators)`
: Decompiles a compiled predicate indicator or a list of compiled predicate indicators returning the entity, entity type, and source level predicate indicators. Requires the entity to be currently loaded.

`execution_context(ExecutionContext, Entity, Sender, This, Self, MetaCallContext, Stack)`
: Allows constructing and accessing execution context components.

`print_message(Kind, Component, Term)`
: Prints a message term after converting it into a list of tokens using the `message_tokens//2` hook non-terminal. When the conversion fails, the message term itself is printed.

`print_message_tokens(Stream, Prefix, Tokens)`
: Prints a list of message tokens to the specified stream and prefixing each line with the specified prefix.

`print_message_token(Stream, Prefix, Token, Tokens)`
: Hook predicate, declared multifile and dynamic, allowing the default printing of a token to be overridden.

`message_tokens(Term, Component)`
: Hook non-terminal, declared multifile and dynamic, allowing the translation of a message into a list of tokens for printing.

`message_prefix_stream(Kind, Component, Prefix, Stream)`
: Hook predicate, declared multifile and dynamic, allowing the definition of line prefix and output stream for messages.

`message_hook(Term, Kind, Component, Tokens)`
: Hook predicate, declared multifile and dynamic, allowing the overriding the default printing of a message.

`trace_event(Event, EventExecutionContext)`
: Hook predicate, declared multifile and dynamic, for handling trace events generated by the execution of source code compiled in debug mode. The Logtalk runtime calls all defined handlers using a failure-driven loop. Thus, care must be taken that the handlers are deterministic to avoid potential termination issues.

`debug_handler_provider(Provider)`
: Multifile predicate for declaring an object that provides a debug handler. There can only be one debug handler provider loaded at the same time. The Logtalk runtime uses this hook predicate for detecting multiple instances of the handler and for better error reporting.

`debug_handler(Event, EventExecutionContext)`
: Multifile predicate for handling debug events generated by the execution of source code compiled in debug mode.

To use these predicates, simply send the corresponding message to the `logtalk` object.

Protocols

Protocols enable the separation between interface and implementation: several objects can implement the same protocol and an object can implement several protocols. Protocols may contain only predicate declarations. In some languages the term *interface* is used with similar meaning. Logtalk allows predicate declarations of any scope within protocols, contrary to some languages that only allow public declarations.

Logtalk defines three built-in protocols, `monitoring`, `expanding`, and `forwarding`, which are described at the end of this section.

Defining a new protocol

We can define a new protocol 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 adapter files. Intermediate Prolog source files (generated by the Logtalk compiler) have, by default, a `_lgt` suffix and a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding adapter 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_lgt.pl` Prolog file (depending on the backend compiler, the names of the intermediate Prolog files may include a directory hash).

Protocol names must be atoms. Objects, categories and protocols share the same name space: we cannot 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([Protocol1, Protocol2, ...])).
    ...
:- end_protocol.
```

In order to maximize protocol reuse, all predicates specified in a protocol should relate to the same functionality. Therefore, the only recommended use of protocol extension is when you need both a minimal protocol and an extended version of the same protocol with additional, useful predicates.

Finding defined protocols

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

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

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 should be either a variable or the name of the new protocol (a Prolog atom, which must 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 define protocol properties and documentation.

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 results in 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 section for details.

Loading files into a protocol

The `include/1` directive can be used to load the contents of a file into a protocol. See the objects section for an example of using this directive.

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 a non-instantiated 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.

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. Depending on the back-end Prolog compiler, a protocol may have additional properties related to the source file where it is defined.

The following protocol properties are supported:

`static`
: The protocol is static

`dynamic`
: The protocol is dynamic (and thus can be abolished in runtime by calling the `abolish_category/1` built-in predicate)

`built_in`
: The protocol is a built-in protocol (and thus always available)

`source_data`
: Source data available for the protocol

`file(Path)`
: Absolute path of the source file defining the protocol (if applicable)

`file(Basename, Directory)`
: Basename and directory of the source file defining the protocol (if applicable)

`lines(BeginLine, EndLine)`
: Source file begin and end lines of the protocol definition (if applicable)

`public(Predicates)`
: List of public predicates declared by the protocol

`protected(Predicates)`
: List of protected predicates declared by the protocol

`private(Predicates)`
: List of private predicates declared by the protocol

`declares(Predicate, Properties)`
: List of properties for a predicate declared by the protocol

`alias(Predicate, Properties)`
: List of properties for a predicate alias declared by the protocol (the properties include `for(Original)`, `from(Entity)`, `non_terminal(NonTerminal)`, and `line_count(Line)` with `Line` being the begin line of the alias directive)

Some of the properties such as line numbers are only available when the protocol is defined in a source file compiled with the `source_data` flag turned on.

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.
```

The same rules applies to protocols implemented by categories.

Built-in protocols

Logtalk defines a set of built-in protocols that are always available for any application.

The built-in protocol *expanding*

Logtalk defines a built-in protocol named `expanding` that contains declarations for the `term_expansion/2` and `goal_expansion/2` predicates. See the description of the `hook` compiler flag for more details.

The built-in protocol *monitoring*

Logtalk defines a built-in protocol named `monitoring` that contains declarations for the `before/3` and `after/3` public event handler predicates. See the event-driven programming section for more details.

The built-in protocol *forwarding*

Logtalk defines a built-in protocol named `forwarding` that contains a declaration for the `forward/1` user-defined message forwarding handler, which is automatically called (if defined) by the runtime for any message that the receiving object does not understand. See also the []/1 control construct.

Categories

Categories are fine-grained units of code reuse and can be regarded as a dual concept of protocols. Categories provide a way to encapsulate a set of related predicate declarations and definitions that do not represent a complete object and that only make sense when composed with other predicates. Categories may also be used to break a complex object in functional units. A category can be imported by several objects (without code duplication), including objects participating in prototype or class-based hierarchies. This concept of categories shares some ideas with Smalltalk-80 functional categories [Goldberg 83], Flavors mix-ins [Moon 86] (without necessarily implying multi-inheritance), and Objective-C categories [Cox 86]. Categories may also *complement* existing objects, thus providing a hot patching mechanism inspired by the Objective-C categories functionality.

Defining a new category

We can define a new category 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 category. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the adapter files. Intermediate Prolog source files (generated by the Logtalk compiler) have, by default, a `_lgt` suffix and a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding adapter file. For instance, we may define a category named `documenting` and save it in a `documenting.lgt` source file that will be compiled to a `documenting_lgt.pl` Prolog file (depending on the backend compiler, the names of the intermediate Prolog files may include a directory hash).

Category names can be atoms or compound terms (when defining parametric categories). Objects, categories, and protocols share the same name space: we cannot have a category with the same name as an object or a protocol.

Category code (directives and predicates) is textually encapsulated by using two Logtalk directives: `category/1-3` and `end_category/0`. The most simple category will be one that is self-contained, not depending on any other Logtalk entity:

```
:- category(Category).
    ...
:- end_category.
```

If a category implements one or more protocols then the opening directive will be:

```
:- category(Category,
    implements([Protocol1, Protocol2, ...])).
    ...
:- end_category.
```

A category may be defined as a composition of other categories by writing:

```
:- category(Category,
    extends([Category1, Category2, ...])).
    ...
:- end_category.
```

This feature should only be used when extending a category without breaking its functional cohesion (for example, when a modified version of a category is needed for importing on several unrelated objects). The preferred way of composing several categories is by importing them into an object. When a category overrides a predicate defined in an extended category, the overridden definition can still be used by using the `alias/2` predicate directive.

Categories cannot inherit from objects. In addition, categories cannot define clauses for dynamic predicates. This restriction applies because a category can be imported by several objects and because we cannot use the database handling built-in methods with categories (messages can only be sent to objects). However, categories may contain declarations for dynamic predicates and they can contain predicates which handle dynamic predicates. For example:

```
:- category(attributes).

    :- public(attribute/2).
    :- public(set_attribute/2).
    :- public(del_attribute/2).

    :- private(attribute_/2).
    :- dynamic(attribute_/2).

    attribute(Attribute, Value) :-
        ::attribute_(Attribute, Value).          % called in the context of "self"

    set_attribute(Attribute, Value) :-
        ::retractall(attribute_(Attribute, _)),  % retracts clauses in "self"
        ::assertz(attribute_(Attribute, Value)). % asserts clause in "self"

    del_attribute(Attribute, Value) :-
        ::retract(attribute_(Attribute, Value)). % retracts clause in "self"

:- end_category.
```

Each object importing this category will have its own `attribute_/2` private, dynamic predicate. The predicates `attribute/2`, `set_attribute/2`, and `del_attribute/2` always access and modify the dynamic predicate contained

in the object receiving the corresponding messages (i.e. *self*). But it's also possible to define predicates that handle dynamic predicates in the context of *this* instead of *self*. For example:

```
:- category(attributes).

    :- public(attribute/2).
    :- public(set_attribute/2).
    :- public(del_attribute/2).

    :- private(attribute_/2).
    :- dynamic(attribute_/2).

    attribute(Attribute, Value) :-
        attribute_(Attribute, Value).           % called in the context of "this"

    set_attribute(Attribute, Value) :-
        retractall(attribute_(Attribute, _)),   % retracts clauses in "this"
        assertz(attribute_(Attribute, Value)).  % asserts clause in "this"

    del_attribute(Attribute, Value) :-
        retract(attribute_(Attribute, Value)).  % retracts clause in "this"

:- end_category.
```

When defining a category that declares and handles dynamic predicates, working in the context of *this* ties those dynamic predicates to the object importing the category while working in the context of *self* allows each object inheriting from the object that imports the category to have its own set of clauses for those dynamic predicates.

Hot patching

A category may explicitly complement one or more existing objects, thus providing hot patching functionality inspired by Objective-C categories:

```
:- category(Category,
    complements([Object1, Object2, ....])).
    ...
:- end_category.
```

This allows us to add missing directives (e.g. to define aliases for complemented object predicates), replace broken predicate definitions, add new predicates, and add protocols and categories to existing objects without requiring access or modifications to their source code. Common scenarios are adding logging or debugging predicates to a set of objects. Complemented objects need to be compiled with the `complements` compiler flag set `allow` (to allow both patching and adding functionality) or `restrict` (to allow only adding new functionality). A complementing category takes preference over a previously loaded complementing category for the same object thus allowing patching a previous patch if necessary.

Note that super calls from predicates defined in complementing categories lookup inherited definitions as if the calls were made from the complemented object instead of the category ancestors. This allows more comprehensive object patching. But it also means that, if you want to patch an object so that it imports a category that extends another category and uses super calls to access the extended category predicates, you will need to define a (possibly empty) complementing category that extends the category that you want to add.

An unfortunate consequence of allowing an object to be patched at runtime using a complementing category is that it disables the use of static binding optimizations for messages sent to the complemented object as it can always be later patched, thus rendering the static binding optimizations invalid.

Another important caveat is that, while a complementing category can replace a predicate definition, local callers of the replaced predicate will still call the unpatched version of the predicate. This is a consequence of the lack of a portable solution at the backend Prolog compiler level for destructively replacing static predicates.

Finding defined categories

We can find, by backtracking, all defined categories by using the `current_category/1` Logtalk built-in predicate with a non-instantiated variable:

```
| ?- current_category(Category).
```

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

Creating a new category in runtime

A category can be dynamically created at runtime by using the `create_category/4` built-in predicate:

```
| ?- create_category(Category, Relations, Directives, Clauses).
```

The first argument should be either a variable or the name of the new category (a Prolog atom, which must not match with an existing entity name). The remaining three arguments correspond to the relations described in the opening category directive and to the category code contents (directives and clauses).

For instance, the call:

```
| ?- create_category(ccc,
                     [implements(ppp)],
                     [private(bar/1)],
                     [(foo(X):-bar(X)), bar(1), bar(2)]).
```

is equivalent to compiling and loading the category:

```
:- category(ccc,
    implements(ppp)).

    :- dynamic.

    :- private(bar/1).

    foo(X) :-
        bar(X).

    bar(1).
    bar(2).

:- end_category.
```

If we need to create a lot of (dynamic) categories at runtime, then is best to 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 category

Dynamic categories can be abolished using the `abolish_category/1` built-in predicate:

```
| ?- abolish_category(Category).
```

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

Category directives

Category directives are used to define category properties, to document a category dependencies on other Logtalk entities, and to load the contents of files into a category.

Dynamic categories

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

```
:- dynamic.
```

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

Category documentation

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

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

See the documenting Logtalk programs section for details.

Loading files into a category

The `include/1` directive can be used to load the contents of a file into a category. See the objects section for an example of using this directive.

Category relationships

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

The built-in predicates `implements_protocol/2-3` and `conforms_to_protocol/2-3` allows us to find which categories implements which protocols:

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

or, if we also want inherited protocols:

```
| ?- conforms_to_protocol(Category, Protocol, Scope).
```

Note that, if we use a non-instantiated variable for the first argument, we will need to use the `current_category/1` built-in predicate to ensure that the returned entity is a category and not an object.

To find which objects import which categories we can use the `imports_category/2` or `imports_category/3` built-in predicates:

```
| ?- imports_category(Object, Category).
```

or, if we want to know the importation scope:

```
| ?- imports_category(Object, Category, Scope).
```

Note that a category may be imported by several objects.

To find which categories extend other categories we can use the `extends_category/2` or `extends_category/3` built-in predicates:

```
| ?- extends_category(Category1, Category2).
```

or, if we want to know the extension scope:

```
| ?- extends_category(Category1, Category2, Scope).
```

Note that a category may be extended by several categories.

To find which categories explicitly complement existing objects we can use the `complements_object/2` built-in predicate:

```
| ?- complements_object(Category, Object).
```

Note that a category may explicitly complement several objects.

Category properties

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

```
| ?- category_property(Category, Property).
```

The following category properties are supported:

`static`
: The category is static

`dynamic`
: The category is dynamic (and thus can be abolished in runtime by calling the `abolish_category/1` built-in predicate)

`built_in`
: The category is a built-in category (and thus always available)

`file(Path)`
: Absolute path of the source file defining the category (if applicable)

`file(Basename, Directory)`
: Basename and directory of the source file defining the category (if applicable)

`lines(BeginLine, EndLine)`
: Source file begin and end lines of the category definition (if applicable)

`events`
: Messages sent from the category generate events

`source_data`
: Source data available for the category

`public(Predicates)`
: List of public predicates declared by the category

`protected(Predicates)`
: List of protected predicates declared by the category

`private(Predicates)`
: List of private predicates declared by the category

`declares(Predicate, Properties)`
: List of properties for a predicate declared by the category

`defines(Predicate, Properties)`
: List of properties for a predicate defined by the category

`includes(Predicate, Entity, Properties)`
: List of properties for an object multifile predicate that are defined in the specified entity (the properties include `number_of_clauses(Number)`, `number_of_rules(Number)`, and `line_count(Line)` with `Line` being the begin line of the multifile predicate clause)

`provides(Predicate, Entity, Properties)`
: List of properties for other entity multifile predicate that are defined in the category (the properties include `number_of_clauses(Number)`, `number_of_rules(Number)`, and `line_count(Line)` with `Line` being the begin line of the multifile predicate clause)

`alias(Predicate, Properties)`
: List of properties for a predicate alias declared by the category (the properties include `for(Original)`, `from(Entity)`, `non_terminal(NonTerminal)`, and `line_count(Line)` with `Line` being the begin line of the alias directive)

`calls(Call, Properties)`
: List of properties for predicate calls made by the category (`Call` is either a predicate indicator or a control construct such as `::/1-2` or `^^/1` with a predicate indicator as argument; note that `Call` may not be ground in case of a call to a control construct where its argument is only know at runtime; the properties include `caller(Caller)`, `alias(Alias)`, and `line_count(Line)` with both `Caller` and `Alias` being predicate indicators and `Line` being the begin line of the predicate clause or directive making the call)

`updates(Predicate, Properties)`
: List of properties for dynamic predicate updates (and also access using the `clause/2` predicate) made by the object (`Predicate` is either a predicate indicator or a control construct such as `::/1-2` or `:/2` with a predicate indicator as argument; note that `Predicate` may not be ground in case of a control construct argument only know at runtime; the properties include `updater(Updater)`, `alias(Alias)`, and `line_count(Line)` with `Updater` being a (possibly multifile) predicate indicator, `Alias` being a predicate indicator, and `Line` being the begin line of the predicate clause or directive updating the predicate)

`number_of_clauses(Number)`
: Total number of predicate clauses defined in the category (includes both user-defined clauses and auxiliary clauses generated by the compiler or by the expansion hooks)

`number_of_rules(Number)`
: Total number of predicate rules defined in the category (includes both user-defined rules and auxiliary rules generated by the compiler or by the expansion hooks)

`number_of_user_clauses(Number)`
: Total number of user-defined predicate clauses defined in the category

`number_of_user_rules(Number)`
: Total number of user-defined predicate rules defined in the category

Some of the properties such as line numbers are only available when the category is defined in a source file compiled with the `source_data` flag turned on.

The properties that return the number of clauses (rules) report the clauses (rules) *textually defined in the object* for both multifile and non-multifile predicates. Thus, these numbers exclude clauses (rules) for multifile predicates *contributed* by other entities.

Importing categories

Any number of objects can import a category. In addition, an object may import any number of categories. The syntax is very simple:

```
:- object(Object,
    imports([Category1, Category2, ...])).
    ...
:- end_object.
```

To make all public predicates imported via a category protected or to make all public and protected predicates private we prefix the category's name with the corresponding keyword:

```
:- object(Object,
    imports(private::Category)).
    ...
:- end_object.
```

or:

```
:- object(Object,
    imports(protected::Category)).
    ...
:- end_object.
```

Omitting the scope keyword is equivalent to writing:

```
:- object(Object,
    imports(public::Category)).
    ...
:- end_object.
```

Calling category predicates

Category predicates can be called from within an object by sending a message to *self* or using a *super* call. Consider the following category:

```
:- category(output).

    :- public(out/1).

    out(X) :-
        write(X), nl.

:- end_category.
```

The predicate `out/1` can be called from within an object importing the category by simply sending a message to *self*. For example:

```
:- object(worker,
    imports(output)).

    ...
    do(Task) :-
        execute(Task, Result),
        ::out(Result).
    ...

:- end_object.
```

This is the recommended way of calling a category predicate that can be specialized/overriden in a descendant object as the predicate definition lookup will start from *self*.

A direct call the predicate definition found in an imported category can be made using the `^^/1` control construct. For example:

```
:- object(worker,
    imports(output)).

    ...
    do(Task) :-
        execute(Task, Result),
        ^^out(Result).
    ...

:- end_object.
```

This alternative should only be used when the user knows a priori that the category predicates will not be specialized or redefined by descendant objects of the object importing the category. Its advantage is that, when the `optimize` compiler flag is turned on, the Logtalk compiler will try to optimize the calls by using static binding. When dynamic binding is used due to e.g. the lack of sufficient information at compilation time, the performance is similar to calling the category predicate using a message to *self* (in both cases a predicate lookup caching mechanism is used).

Parametric categories

Category predicates can be parameterized in the same way as object predicates by using a compound term as the category identifier. The category parameters can be accessed by calling the `parameter/2` or `this/1` built-in local methods in the category predicate clauses or by using *parameter variables*. Category parameter values can be defined by the importing objects. For example:

```
:- object(speech(Season, Event),
    imports([dress(Season), speech(Event)])).
    ...
:- end_object.
```

Note that access to category parameters is only possible from within the category. In particular, calls to the `this/1` built-in local method from category predicates always access the importing object identifier (and thus object parameters, not category parameters).

Predicates

Predicate directives and clauses can be encapsulated inside objects and categories. Protocols can only contain predicate directives. From the point-of-view of an object-oriented language, predicates allows both object state and object behavior to be represented. Mutable object state can be represented using dynamic object predicates.

Reserved predicate names

For performance reasons, a few predicates have a fixed interpretation. These predicates are declared in the built-protocols. They are: `goal_expansion/2` and `term_expansion/2`, declared in the `expanding` protocol; `before/3` and `after/3`, declared in the `monitoring` protocol; and `forward/1`, declared in the `forwarding` protocol. By default, the compiler prints a warning when a definition for one of these predicates are found but the reference to the corresponding built-in protocol is missing.

Declaring predicates

All object (or category) predicates that we want to access from other objects must be explicitly declared. A predicate declaration must contain, at least, a scope directive. Other directives may be used to document the predicate or to ensure proper compilation of the predicate definitions.

Scope directives

A predicate scope directive specifies *from where* the predicate can be called, i.e. its *visibility*. Predicates can be *public*, *protected*, *private*, or *local*. Public predicates can be called from any object. Protected predicates can only be called from the container object or from a container descendant. Private predicates can only be called from the container object. Local predicates, like private predicates, can only be called from the container object (or category) but they are *invisible* to the reflection built-in methods (`current_op/3`, `current_predicate/1`, and `predicate_property/2`) and to the message error handling mechanisms (i.e. sending a message corresponding to a local predicate results in a `predicate_declaration` existence error, not in a scope error).

The scope declarations are made using the directives `public/1`, `protected/1`, and `private/1`. For example:

```
:- public(init/1).

:- protected(valid_init_option/1).

:- private(process_init_options/1).
```

If a predicate does not have a scope declaration, it is assumed that the predicate is local. Note that we do not need to write scope declarations for all defined predicates. One exception is local dynamic predicates: declaring them as private predicates may allow the Logtalk compiler to generate optimized code for asserting and retracting clauses.

Note that a predicate scope directive doesn't specify *where* a predicate is, or can be, defined. For example, a private predicate can only be called from an object holding its scope directive. But it can be defined in descendant objects. A typical example is an object playing the role of a class defining a private (possibly dynamic) predicate for its descendant instances. Only the class can call (and possibly assert/retract clauses for) the predicate but its clauses can be found/defined in the instances themselves.

Mode directive

Often predicates can only called using specific argument patterns. The valid arguments and instantiation modes of those arguments can be documented by using the `mode/2` directive. For example:

```
:- mode(member(?term, ?list), zero_or_more).
```

The first directive argument describes a valid calling mode. The minimum information will be the instantiation mode of each argument. The first four possible values are described in [ISO 95]). The remaining two can be found in use in some Prolog systems.

`+`
: Argument must be instantiated (but not necessarily ground).

`-`
: Argument should be a free (non-instantiated) variable (when bound, the call will unify the returned term with the given term).

`?`
: Argument can either be instantiated or free.

`@`
: Argument will not be further instantiated (modified).

`++`
: Argument must be ground.

`--`
: Argument must be unbound.

These four mode atoms are also declared as prefix operators by the Logtalk compiler. This makes it possible to include type information for each argument like in the example above. Some of the possible type values are: `event`, `object`, `category`, `protocol`, `callable`, `term`, `nonvar`, `var`, `atomic`, `atom`, `number`, `integer`, `float`, `compound`, and `list`. The first four are Logtalk specific. The remaining are common Prolog types. We can also use our own types that can be either atoms or ground compound terms.

The second directive argument documents the number of proofs (not necessarily distinct solutions) for the specified mode. Note that different modes for the same predicate often have different determinism. The possible values are:

`zero`
: Predicate always fails.

`one`
: Predicate always succeeds once.

`zero_or_one`
: Predicate either fails or succeeds.

`zero_or_more`
: Predicate has zero or more proofs.

`one_or_more`
: Predicate has one or more proofs.

`one_or_error`
: Predicate either succeeds once or throws an error (see below).

`error`
: Predicate will throw an error (see below).

Mode declarations can also be used to document that some call modes will throw an error. For instance, regarding the `arg/3` ISO Prolog built-in predicate, we may write:

```
:- mode(arg(-, -, +), error).
```

Note that most predicates have more than one valid mode implying several mode directives. For example, to document the possible use modes of the `atom_concat/3` ISO built-in predicate we would write:

```
:- mode(atom_concat(?atom, ?atom, +atom), one_or_more).
:- mode(atom_concat(+atom, +atom, -atom), zero_or_one).
```

Some old Prolog compilers supported some sort of mode directives to improve performance. To the best of my knowledge, there is no modern Prolog compiler supporting these kind of directive. The current version of the Logtalk compiler just parses and than discards this directive (however, see the description on synchronized predicates on the multi-threading programming section). Nevertheless, the use of mode directives is a good starting point for documenting your predicates.

Meta-predicate directive

Some predicates may have arguments that will be called as goals or closures that will be used for constructing a goal. To ensure that these goals will be executed in the correct scope (i.e. in the *calling context*, not in the meta-predicate *definition context*) we need to use the `meta_predicate/1` directive. For example:

```
:- meta_predicate(findall(*, 0, *)).
```

The meta-predicate mode arguments in this directive have the following meaning:

`0`
: Meta-argument that will be called as a goal.

`N`
: Meta-argument that will be a closure used to construct a call by appending `N` arguments. The value of `N` must be a non-negative integer.

`::`
: Argument that is context-aware but that will not be called as a goal.

`^`
: Goal that may be existentially quantified (`Vars^Goal`).

`*`
: Normal argument.

The following meta-predicate mode arguments are for use only when writing backend Prolog adapter files to deal with proprietary built-in meta-predicates and meta-directives:

`/`
: Predicate indicator (`Name/Arity`), list of predicate indicators, or conjunction of predicate indicators.

`//`
: Non-terminal indicator (`Name//Arity`), list of predicate indicators, or conjunction of predicate indicators.

`[0]`
: List of goals.

`[N]`
: List of closures.

`[/]`
: List of predicate indicators.

`[//]`
: List of non-terminal indicators.

To the best of my knowledge, the use of non-negative integers to specify closures has first introduced on Quintus Prolog for providing information for predicate cross-reference tools.

As each Logtalk entity is independently compiled, this directive must be included in every object or category that contains a definition for the described meta-predicate, even if the meta-predicate declaration is inherited from another entity, to ensure proper compilation of meta-arguments.

Discontiguous directive

The clause of an object (or category) predicate may not be contiguous. In that case, we must declare the predicate discontiguous by using the `discontiguous/1` directive:

```
:- discontiguous(foo/1).
```

This is a directive that we should avoid using: it makes your code harder to read and it is not supported by some Prolog compilers.

As each Logtalk entity is compiled independently from other entities, this directive must be included in every object or category that contains a definition for the described predicate (even if the predicate declaration is inherited from other entity).

Dynamic directive

An object predicate can be static or dynamic. By default, all object predicates are static. To declare a dynamic predicate we use the `dynamic/1` directive:

```
:- dynamic(foo/1).
```

This directive may also be used to declare dynamic grammar rule non-terminals. As each Logtalk entity is compiled independently from other entities, this directive must be included in every object that contains a definition for the described predicate (even if the predicate declaration is inherited from other object or imported from a category). If we omit the dynamic declaration then the predicate definition will be compiled static. In the case of dynamic objects, static predicates cannot be redefined using the database built-in methods (despite being internally compiled to dynamic code).

Dynamic predicates can be used to represent persistant mutable object state. Note that static objects may declare and define dynamic predicates.

Operator directive

An object (or category) predicate can be declared as an operator using the familiar `op/3` directive:

```
:- op(Priority, Specifier, Operator).
```

Operators are local to the object (or category) where they are declared. This means that, if you declare a public predicate as an operator, you cannot use operator notation when sending to an object (where the predicate is visible) the respective message (as this would imply visibility of the operator declaration in the context of the *sender* of the message). If you want to declare global operators and, at the same time, use them inside an entity, just write the corresponding directives at the top of your source file, before the entity opening directive.

When the same operators are used on several entities within the same source file, the corresponding directives must appear before any entity that uses them. However, this results in a global scope for the operators. If you prefer the operators to be local to the source file, just *undefine* them at the end of the file. For example:

```
:- op(400, xfx, results).  % before any entity that uses the operator

...

:- op(0, xfx, results).    % after all entities that used the operator
```

Uses directive

When a predicate makes heavy use of predicates defined on other objects, its predicate clauses can be verbose due to all the necessary message sending goals. Consider the following example:

```
foo :-
    ...,
    findall(X, list::member(X, L), A),
    list::append(A, B, C),
    list::select(Y, C, R),
    ...
```

Logtalk provides a directive, `uses/2`, which allows us to simplify the code above. The usage template for this directive is:

```
:- uses(Object, [Functor1/Arity1, Functor2/Arity2, ...]).
```

Rewriting the code above using this directive results in a simplified and more readable predicate definition:

```
:- uses(list, [
    append/3, member/2, select/3
]).

foo :-
    ...,
    findall(X, member(X, L), A),
    append(A, B, C),
    select(Y, C, R),
    ...
```

Logtalk also supports an extended version of this directive that allows the declaration of predicate alias using the notation `Predicate as Alias` (or the alternative notation `Predicate::Alias`). For example:

```
:- uses(btrees, [new/1 as new_btree/1]).
:- uses(queues, [new/1 as new_queue/1]).
```

You may use this extended version for solving conflicts between predicates declared on several `uses/2` directives or just for giving new names to the predicates that will be more meaningful on their using context.

The `uses/2` directive allows simpler predicate definitions as long as there are no conflicts between the predicates declared in the directive and the predicates defined in the object (or category) containing the directive. A predicate (or its alias if defined) cannot be listed in more than one `uses/2` directive. In addition, a `uses/2` directive cannot list a predicate (or its alias if defined) which is defined in the object (or category) containing the directive. Any conflicts are reported by Logtalk as compilation errors.

Alias directive

Logtalk allows the definition of an alternative name for an inherited or imported predicate (or for an inherited or imported grammar rule non-terminal) through the use of the `alias/2` directive:

```
:- alias(Entity, [Predicate1 as Alias1, Predicate2 as Alias2, ...]).
```

This directive can be used in objects, protocols, or categories. The first argument, `Entity`, must be an entity referenced in the opening directive of the entity containing the `alias/2` directive. It can be an implemented protocol, an imported category, an extended prototype, an instantiated class, or a specialized class. The second argument is a list of pairs of predicate indicators (or grammar rule non-terminal indicators) using the `as` infix operator as connector.

A common use for the `alias/2` directive is to give an alternative name to an inherited predicate in order to improve readability. For example:

```
:- object(square,
    extends(rectangle)).

    :- alias(rectangle, [width/1 as side/1]).

    ...

:- end_object.
```

The directive allows both `width/1` and `side/1` to be used as messages to the object `square`. Thus, using this directive, there is no need to explicitly declare and define a "new" `side/1` predicate. Note that the `alias/2` directive does not rename a predicate, only provides an alternative, additional name; the original name continues to be available (although it may be masked due to the default inheritance conflict mechanism).

Another common use for this directive is to solve conflicts when two inherited predicates have the same functor and arity. We may want to call the predicate which is masked out by the Logtalk lookup algorithm (see the Inheritance section) or we may need to call both predicates. This is simply accomplished by using the `alias/2` directive to give alternative names to masked out or conflicting predicates. Consider the following example:

```
:- object(my_data_structure,
    extends(list, set)).

    :- alias(list, [member/2 as list_member/2]).
    :- alias(set,  [member/2 as set_member/2]).

    ...

:- end_object.
```

Assuming that both `list` and `set` objects define a `member/2` predicate, without the `alias/2` directives, only the definition of `member/2` predicate in the object `list` would be visible on the object `my_data_structure`, as a result of the application of the Logtalk predicate lookup algorithm. By using the `alias/2` directives, all the following messages would be valid (assuming a public scope for the predicates):

```
| ?- my_data_structure::list_member(X, L).    % uses list member/2

| ?- my_data_structure::set_member(X, L).     % uses set member/2

| ?- my_data_structure::member(X, L).         % uses list member/2
```

When used this way, the `alias/2` directive provides functionality similar to programming constructs of other object-oriented languages which support multi-inheritance (the most notable example probably being the renaming of inherited features in Eiffel).

Note that the `alias/2` directive never hides a predicate which is visible on the entity containing the directive as a result of the Logtalk lookup algorithm. However, it may be used to make visible a predicate which otherwise would be masked by another predicate, as illustrated in the above example.

The `alias/2` directive may also be used to give access to an inherited predicate, which otherwise would be masked by another inherited predicate, while keeping the original name as follows:

```
:- object(my_data_structure,
    extends(list, set)).

    :- alias(list, [member/2 as list_member/2]).
    :- alias(set,  [member/2 as set_member/2]).

    member(X, L) :-
        ::set_member(X, L).

    ...

:- end_object.
```

Thus, when sending the message `member/2` to `my_data_structure`, the predicate definition in `set` will be used instead of the one contained in `list`.

Documenting directive

A predicate can be documented with arbitrary user-defined information by using the `info/2` directive:

```
:- info(Name/Arity, List).
```

The second argument is a list of `Key is Value` terms. See the Documenting Logtalk programs section for details.

Multifile directive

A predicate can be declared *multifile* by using the `multifile/1` directive:

```
:- multifile(Name/Arity).
```

This allows clauses for a predicate to be defined in several objects and/or categories. This is a directive that should be used with care. Support for this directive have been added to Logtalk primarily to support migration of Prolog module code. Spreading clauses for a predicate among several Logtalk entities can be handy in some cases but can also make your code difficult to understand. Logtalk precludes using a multifile predicate for breaking object encapsulation by checking that the object (or category) declaring the predicate (using a scope directive) defines it also as multifile. This entity is said to contain the *primary declaration* for the multifile predicate. In addition, note that the `multifile/1` directive is mandatory when defining multifile predicates.

Consider the following simple example:

```
:- object(main).

    :- public(a/1).
    :- multifile(a/1).
    a(1).

:- end_object.
```

After compiling and loading the `main` object, we can define other objects (or categories) that contribute with clauses for the multifile predicate. For example:

```
:- object(other).

    :- multifile(main::a/1).
    main::a(2).
    main::a(X) :-
        b(X).

    b(3).
    b(4).

:- end_object.
```

After compiling and loading the above objects, you can use queries such as:

```
| ?- main::a(X).

X = 1 ;
X = 2 ;
X = 3 ;
X = 4
yes
```

Entities containing primary multifile predicate declarations must always be compiled before entities defining clauses for those multifile predicates. The Logtalk compiler will print a warning if the scope directive is missing.

Multifile predicates may also be declared dynamic using the same `Entity::Name/Arity` notation (multifile predicates are static by default).

When a clause of a multifile predicate is a rule, its body is compiled within the context of the object or category defining the clause. This allows clauses for multifile predicates to call local object or category predicates. But the values of the *sender*, *this*, and *self* in the implicit execution context are passed from the clause head to the clause body. This is necessary to ensure that these values are always valid and to allow multifile predicate clauses to be defined in categories. A call to the `parameter/2` execution context methods, however, retrieves parameters of the entity defining the clause, not from the entity for which the clause is defined. The parameters of the entity for which the clause is defined can be accessed by simple unification at the clause head.

Local calls to the database methods from multifile predicate clauses defined in an object take place in the object own database instead of the database of the entity holding the multifile predicate primary declaration. Similarly, local calls to the `expand_term/2` and `expand_goal/2` methods from a multifile predicate clause look for clauses of the `term_expansion/2` and `goal_expansion/2` hook predicates starting from the entity defining the clause instead of the entity holding the multifile predicate primary declaration. Local calls to the `current_predicate/1`, `predicate_property/2`, and `current_op/3` methods from multifile predicate clauses defined in an object also lookup predicates and their properties in the object own database instead of the database of the entity holding the multifile predicate primary declaration.

Coinductive directive

A predicate can be declared *coinductive* by using the `coinductive/1` directive. For example:

```
:- coinductive(comember/2).
```

Logtalk support for coinductive predicates is experimental and requires a back-end Prolog compiler with minimal support for cyclic terms.

Defining predicates

Object predicates

We define object predicates as we have always defined Prolog predicates, the only difference be that we have four more control structures (the three message sending operators plus the external call operator) to play with. For example, if we wish to define an object containing common utility list predicates like `append/2` or `member/2` we could write something like:

```
:- object(list).

    :- public(append/3).
    :- public(member/2).

    append([], L, L).
    append([H| T], L, [H| T2]) :-
        append(T, L, T2).

    member(H, [H| _]).
    member(H, [_| T]) :-
        member(H, T).

:- end_object.
```

Note that, abstracting from the opening and closing object directives and the scope directives, what we have written is plain Prolog. Calls in a predicate definition body default to the local predicates, unless we use the message sending operators or the external call operator. This enables easy conversion from Prolog code to Logtalk objects: we just need to add the necessary encapsulation and scope directives to the old code.

Category predicates

Because a category can be imported by multiple objects, dynamic private predicates must be called either in the context of *self*, using the *message to self* control structure, `::/1`, or in the context of *this* (i.e. in the context of the object importing

the category). For example, if we want to define a category implementing variables using destructive assignment where the variable values are stored in *self* we could write:

```
:- category(variable).

    :- public(get/2).
    :- public(set/2).

    :- private(value_/2).
    :- dynamic(value_/2).

    get(Var, Value) :-
        ::value_(Var, Value).

    set(Var, Value) :-
        ::retractall(value_(Var, _)),
        ::asserta(value_(Var, Value).

:- end_category.
```

In this case, the `get/2` and `set/2` predicates will always access/update the correct definition, contained in the object receiving the messages. The alternative, storing the variable values in *this*, such that each object importing the category will have its own definition for the `value_/2` private predicate is simple: just omit the use of the `::/1` control construct in the code above.

A category can only contain clauses for static predicates. Nevertheless, as the example above illustrates, there are no restrictions in declaring and calling dynamic predicates from inside a category.

Meta-predicates

Meta-predicates may be defined inside objects (and categories) as any other predicate. A meta-predicate is declared using the `meta_predicate/1` directive as described earlier on this section. When defining a meta-predicate, the arguments in the clause heads corresponding to the meta-arguments must be variables. All meta-arguments are called in the context of the entity calling the meta-predicate.

Some meta-predicates have meta-arguments which are not goals but closures. Logtalk supports the definition of meta-predicates that are called with closures instead of goals as long as the definition uses the Logtalk built-in predicate `call/N` to call the closure with the additional arguments. For example:

```
:- public(all_true/2).
:- meta_predicate(all_true(1, *)).

all_true(_, []).
all_true(Closure, [Arg| Args]) :-
    call(Closure, Arg),
    all_true(Closure, Args).
```

Note that in this case the meta-predicate directive specifies that the closure will be extended with exactly one extra argument.

When calling a meta-predicate, a closure can correspond to a user-defined predicate, a built-in predicate, a lambda expression, or a control construct.

Lambda expressions

The use of lambda expressions as meta-predicate goal and closure arguments often saves writing auxiliary predicates for the sole purpose of calling the meta-predicates. A simple example of a lambda expression is:

```
| ?- meta::map([X,Y]>>(Y is 2*X), [1,2,3], Ys).
Ys = [2,4,6]
yes
```

In this example, a lambda expression, `[X,Y]>>(Y is 2*X)`, is used as an argument to the `map/3` list mapping predicate, defined in the library object `meta`, in order to double the elements of a list of integers. Using a lambda expression avoids writing an auxiliary predicate for the sole purpose of doubling the list elements. The lambda parameters are represented by the list `[X,Y]`, which is connected to the lambda goal, `(Y is 2*X)`, by the `(>>)/2` operator.

Currying is supported. I.e. it is possible to write a lambda expression whose goal is another lambda expression. The above example can be rewritten as:

```
| ?- meta::map([X]>>([Y]>>(Y is 2*X)), [1,2,3], Ys).
Ys = [2,4,6]
yes
```

Lambda expressions may also contain lambda free variables. I.e. variables that are global to the lambda expression. For example, using GNU Prolog as the back-end compiler, we can write:

```
| ?- meta::map({Z}/[X,Y]>>(Z#=X+Y), [1,2,3], Zs).
Z = _#22(3..268435455)
Zs = [_#3(2..268435454),_#66(1..268435453),_#110(0..268435452)]
yes
```

The ISO Prolog construct `{}/1` for representing the lambda free variables as this representation is often associated with set representation. Note that the order of the free variables is of no consequence (on the other hand, a list is used for the lambda parameters as their order does matter).

Both lambda free variables and lambda parameters can be any Prolog term. Consider the following example by Markus Triska:

```
| ?- meta::map([A-B,B-A]>>true, [1-a,2-b,3-c], Zs).
Zs = [a-1,b-2,c-3]
yes
```

Lambda expressions can be used, as expected, in non-deterministic queries as in the following example using SWI-Prolog as the back-end compiler and Markus Triska's CLP(FD) library:

```
| ?- meta::map({Z}/[X,Y]>>(clpfd:(Z#=X+Y)), Xs, Ys).
Xs = [],
Ys = [] ;
Xs = [_G1369],
Ys = [_G1378],
_G1369+_G1378#=Z ;
Xs = [_G1579, _G1582],
Ys = [_G1591, _G1594],
_G1582+_G1594#=Z,
_G1579+_G1591#=Z ;
Xs = [_G1789, _G1792, _G1795],
Ys = [_G1804, _G1807, _G1810],
_G1795+_G1810#=Z,
_G1792+_G1807#=Z,
_G1789+_G1804#=Z ;
...
```

As illustrated by the above examples, lambda expression syntax reuses the ISO Prolog construct `{}/1` and the standard operators `(/)/2` and `(>>)/2`, thus avoiding defining new operators, which is always tricky for a portable system such as Logtalk. The operator `(>>)/2` was chosen as it suggests an arrow, similar to the syntax used in other languages such as OCaml and Haskell to connect lambda parameters with lambda functions. This syntax was also chosen in order to simplify parsing, error checking, and compilation of lambda expressions. The full specification of the lambda expression syntax can be found in the reference manual.

The compiler checks whenever possible that all variables in a lambda expression are either classified as free variables or as lambda parameters. The use of non-classified variables in a lambda expression should be regarded as a programming error. Unfortunately, the dynamic features of the language and lack of sufficient information at compile time may prevent the compiler of checking all uses of lambda expressions. The compiler also checks if a variable is classified as both a free variable and a lambda parameter. An optimizing meta-predicate and lambda expression compiler, based on the term-expansion mechanism, is provided for practical performance.

Definite clause grammar rules

Definite clause grammar rules provide a convenient notation to represent the rewrite rules common of most grammars in Prolog. In Logtalk, definite clause grammar rules can be encapsulated in objects and categories. Currently, the ISO/IEC WG17 group is working on a draft specification for a definite clause grammars Prolog standard. Therefore, in the mean time, Logtalk follows the common practice of Prolog compilers supporting definite clause grammars, extending it to

support calling grammar rules contained in categories and objects. A common example of a definite clause grammar is the definition of a set of rules for parsing simple arithmetic expressions:

```
:- object(calculator).

    :- public(parse/2).

    parse(Expression, Value) :-
        phrase(expr(Value), Expression).

    expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
    expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
    expr(X) --> term(X).

    term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
    term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
    term(Z) --> number(Z).

    number(C) --> "+", number(C).
    number(C) --> "-", number(X), {C is -X}.
    number(X) --> [C], {0'0 =< C, C =< 0'9, X is C - 0'0}.

:- end_object.
```

The predicate `phrase/2` called in the definition of predicate `parse/2` above is a Logtalk built-in method, similar to the predicate with the same name found on most Prolog compilers that support definite clause grammars. After compiling and loading this object, we can test the grammar rules with calls such as the following one:

```
| ?- calculator::parse("1+2-3*4", Result).

Result = -9
yes
```

In most cases, the predicates resulting from the translation of the grammar rules to regular clauses are not declared. Instead, these predicates are usually called by using the built-in methods `phrase/2` and `phrase/3` as shown in the example above. When we want to use the built-in methods `phrase/2` and `phrase/3`, the non-terminal used as first argument must be within the scope of the *sender*. For the above example, assuming that we want the predicate corresponding to the `expr//1` non-terminal to be public, the corresponding scope directive would be:

```
:- public(expr//1).
```

The `//` infix operator used above tells the Logtalk compiler that the scope directive refers to a grammar rule non-terminal, not to a predicate. The idea is that the predicate corresponding to the translation of the `expr//1` non-terminal will have a number of arguments equal to one plus the number of additional arguments necessary for processing the subjacent lists of tokens.

In the body of a grammar rule, we can call rules that are inherited from ancestor objects, imported from categories, or contained in other objects. This is accomplished by using non-terminals as messages. Using a non-terminal as a message to *self* allows us to call grammar rules in categories and ancestor objects. To call grammar rules encapsulated in other

objects, we use a non-terminal as a message to those objects. Consider the following example, containing grammar rules for parsing natural language sentences:

```
:- object(sentence,
    imports(determiners, nouns, verbs)).

    :- public(parse/2).

    parse(List, true) :-
        phrase(sentence, List).
    parse(_, false).

    sentence --> noun_phrase, verb_phrase.

    noun_phrase --> ::determiner, ::noun.
    noun_phrase --> ::noun.

    verb_phrase --> ::verb.
    verb_phrase --> ::verb, noun_phrase.

:- end_object.
```

The categories imported by the object would contain the necessary grammar rules for parsing determiners, nouns, and verbs. For example:

```
:- category(determiners).

    :- private(determiner//0).

    determiner --> [the].
    determiner --> [a].

:- end_category.
```

Along with the message sending operators (`::/1`, `::/2`, and `^^/1`), we may also use other control constructs such as `\+/1`, `!/0`, `;/2`, `->/2`, and `{}/1` in the body of a grammar. In addition, grammar rules may contain meta-calls (a variable taking the place of a non-terminal), which are translated to calls of the built-in method `phrase/3`.

You may have noticed that Logtalk defines `{}/1` as a control construct for bypassing the compiler when compiling a clause body goal. As exemplified above, this is the same control construct that is used in grammar rules for bypassing the expansion of rule body goals when a rule is converted into a clause. Both control constructs can be combined in order to

call a goal from a grammar rule body, while bypassing at the same time the Logtalk compiler. Consider the following example:

```
bar :-
    write('bar predicate called'), nl.

:- object(bypass).

    :- public(foo//0).

    foo --> {{bar}}.

:- end_object.
```

After compiling and loading this code, we may try the following query:

```
| ?- logtalk << phrase(bypass::foo, _, _).

bar predicate called
yes
```

This is the expected result as the expansion of the grammar rule into a clause leaves the `{bar}` goal untouched, which, in turn, is converted into the goal `bar` when the clause is compiled.

A grammar rule non-terminal may be declared as dynamic or discontiguous, as any object predicate, using the same *`Name//Arity`* notation illustrated above for the scope directives. In addition, grammar rule non-terminals can be documented using the `info/2` directive, as in the following example:

```
:- public(sentence//0).

:- info(sentence//0, [
    comment is 'Rewrites a sentence into a noun phrase and a verb phrase.']).
```

Built-in object predicates (methods)

Logtalk defines a set of built-in object predicates or methods to access message execution context, to find sets of solutions, to inspect objects, for database handling, for term and goal expansion, and for printing messages. Similar to Prolog built-in predicates, these built-in methods should not be redefined.

Execution context methods

Logtalk defines five built-in private methods to access an object execution context. These methods are in the common usage scenarios translated to a single unification performed at compile time with a clause head context argument. Therefore, they can be freely used without worrying about performance penalties. When called from inside a category, these methods refer to the execution context of the object importing the category. These methods are private and cannot be used as messages to objects.

To find the object that received the message under execution we may use the `self/1` method. We may also retrieve the object that has sent the message under execution using the `sender/1` method.

The method `this/1` enables us to retrieve the name of the object for which the predicate clause whose body is being executed is defined instead of using the name directly. This helps to avoid breaking the code if we decide to change the object name and forget to change the name references. This method may also be used from within a category. In this case, the method returns the object importing the category on whose behalf the predicate clause is being executed.

Here is a short example including calls to these three object execution context methods:

```
:- object(test).

    :- public(test/0).

    test :-
        this(This),
        write('Calling a predicate definition contained in '), writeq(This), nl,
        self(Self),
        write('to answer a message received by '), writeq(Self), nl,
        sender(Sender),
        write('that was sent by '), writeq(Sender), nl, nl.

:- end_object.

:- object(descendant,
    extends(test)).

:- end_object.
```

After compiling and loading these two objects, we can try the following goal:

```
| ?- descendant::test.

Calling a predicate definition contained in test
to answer a message received by descendant
that was sent by user
yes
```

Note that the goals `self(Self)`, `sender(Sender)`, and `this(This)`, being translated to unifications with the clause head context arguments at compile time, are effectively removed from the clause body. Therefore, a clause such as:

```
predicate(Arg) :-
    self(Self),
    atom(Arg),
    ... .
```

is compiled with the goal `atom(Arg)` as the first condition on the clause body. As such, the use of these context execution methods do not interfere with the optimizations that some Prolog compilers perform when the first clause body condition is a call to a built-in type-test predicate or a comparison operator.

For parametric objects and categories, the method `parameter/2` enables us to retrieve current parameter values (see the section on parametric objects for a detailed description). For example:

```
:- object(block(_Color)).

    :- public(test/0).

    test :-
        parameter(1, Color),
        write('Color parameter value is '), writeq(Color), nl.

:- end_object.
```

After compiling and loading these two objects, we can try the following goal:

```
| ?- block(blue)::test.

Color parameter value is blue
yes
```

Calls to the `parameter/2` method are translated to a compile time unification when the second argument is a variable. When the second argument is bound, the calls are translated to a call to the built-in predicate `arg/3`.

When type-checking predicate arguments, it is often useful to include the predicate execution context when reporting an argument error. The `context/1` method provides access to that context. For example, assume a predicate `foo/2` that takes an atom and an integer as arguments. We could type-check the arguments by writing (using the library `type` object):

```
foo(A, N) :-
    % type-check arguments
    context(Context),
    type::check(atom, A, Context),
    type::check(integer, N, Context),
    % arguments are fine; go ahead
    ... .
```

Error handling and throwing methods

Besides the `catch/3` and `throw/1` methods inherited from Prolog, Logtalk also provides a set of convenience methods to throw standard `error/2` exception terms: `instantiation_error/0`, `type_error/2`, `domain_error/2`, `existence_error/2`, `permission_error/3`, `representation_error/1`, `evaluation_error/1`, and `resource_error/1`.

Database methods

Logtalk provides a set of built-in methods for object database handling similar to the usual database Prolog predicates: `abolish/1`, `asserta/1`, `assertz/1`, `clause/2`, `retract/1`, and `retractall/1`. These methods always operate on the database of the object receiving the corresponding message.

When working with dynamic grammar rule non-terminals, you may use the built-in method `expand_term/2` convert a grammar rule into a clause that can than be used with the database methods.

Meta-call methods

Logtalk supports the generalized `call/1-N` meta-predicate. This built-in private meta-predicate must be used in the implementation of meta-predicates which work with closures instead of goals. In addition, Logtalk supports the built-in private meta-predicates `ignore/1`, `once/1`, and `\+/1`. These methods cannot be used as messages to objects.

All solutions methods

The usual all solutions meta-predicates are built-in private methods in Logtalk: `bagof/3`, `findall/3`, `findall/4`, and `setof/3`. There is also a `forall/2` method that implements generate and test loops. These methods cannot be used as messages to objects.

Reflection methods

Logtalk provides a comprehensive set of built-in predicates and built-in methods for querying about entities and predicates. Some of information, however, requires that the source files are compiled with the `source_data` flag turned on.

The reflection API supports two different views on entities and their contents, which we may call the *transparent box view* and the *black box view*. In the transparent box view, we look into an entity disregarding how it will be used and returning all information available on it, including predicate declarations and predicate definitions. This view is supported by the entity property built-in predicates. In the black box view, we look into an entity from an usage point-of-view using built-in methods for inspecting object operators and predicates that are within scope from where we are making the call: `current_op/3`, which returns operator specifications, `predicate_property/2`, which returns predicate properties, and `current_predicate/1`, which enables us to query about predicate definitions. See below for a more detailed description of these methods.

Definite clause grammar parsing methods and non-terminals

Logtalk supports two definite clause grammar parsing built-in private methods, `phrase/2` and `phrase/3`, with definitions similar to the predicates with the same name found on most Prolog compilers that support definite clause grammars. These methods cannot be used as messages to objects.

Logtalk also supports `phrase//1`, `call//1-N`, and `eos//0` built-in non-terminals. The `call//1-N` non-terminals takes a closure (which can be a lambda expression) plus zero or more additional arguments and are processed by appending the input list of tokens and the list of remaining tokens to the arguments.

Term and goal expansion methods

Logtalk supports a `expand_term/2` built-in method for expanding a term into another term or a list of terms. This method is mainly used to translate grammar rules into Prolog clauses. It can be customized, e.g. for bypassing the default Logtalk grammar rule translator, by defining clauses for the `term_expansion/2` hook predicate. Logtalk also supports a `expand_goal/2` built-in method for expanding a goal. This method can also be customized by defining clauses for the `goal_expansion/2` hook predicate.

Term and goal expansion may be performed either by calling the `expand_term/2` and `expand_goal/2` built-in methods explicitly or by using *hook objects*. An hook object is simply an object defining clauses for the term- and goal-expansion hook predicates. To compile a source file using a hook object for expanding its terms and goals, you can use the `hook/1` compiler flag in the second argument of the `logtalk_compile/2` or `logtalk_load/2` built-in predicates. In alternative, you can use a `set_logtalk_flag/2` directive in the source file itself. When compiling a source file, the compiler will first try the source file specific hook object, if defined. If that fails, it tries the default hook object, if defined. If that also fails, the compiler tries the Prolog dialect specific expansion predicate definitions if defined in the adapter file.

Sometimes we have multiple hook objects that we need to use in the compilation of a source file. The Logtalk library includes support for two basic expansion workflows: a pipeline of hook objects, where the expansion results from an hook object are feed to the next hook object in the pipeline, and a set of hook objects, where expansions are tried until one of is found that succeeds. These workflows are implemented as parametric objects allowing combining them to implement more sophisticated expansion workflows.

Clauses for the `term_expansion/2` and `goal_expansion/2` predicates defined within an object or a category are never used in the compilation of the object or the category itself, however. In order to use clauses for the `term_expansion/2` and `goal_expansion/2` predicates defined in plain Prolog, simply specify the pseudo-object `user` as the hook object when compiling source files. When using backend Prolog compilers that support a module system, it can also be specified a module containing clauses for the expanding predicates as long as the module name doesn't coincide with an object name.

Logtalk provides a `logtalk_load_context/2` built-in predicate that can be used to access the compilation/loading context when performing term-expansion or goal-expansion.

Printing messages

Logtalk features a *structured message printing* mechanism. This mechanism gives the programmer full control of message printing, allowing it to filter, rewrite, or redirect any message. The origins of the message printing mechanism that inspired the Logtalk implementation goes back to Quintus Prolog, where it was apparently implemented by Dave Bowen (thanks to Richard O'Keefe for the historical bits). Variations of this mechanism can also be found currently on e.g. SICStus Prolog, SWI-Prolog, and YAP.

Why a mechanism for printing messages? Consider the different components in a Logtalk application development and execution. At the bottom level, you have the Logtalk compiler and runtime. The Logtalk compiler writes messages related to e.g. compiling and loading files, compiling entities, compilation warnings and errors. The Logtalk runtime may write banner messages and handles execution errors that may result in printing human-level messages. The development environment can be console-based or you may be using a GUI tool such as PDT. In the latter case, PDT needs to intercept the Logtalk compiler and runtime messages to present the relevant information using its GUI. Then you have all the other components in a typical application. For example, your own libraries and third-party libraries. The libraries may want to print messages on its own, e.g. banners, debugging information, or logging information. As you assemble all your application components, you want to have the final word on which messages are printed, where, an in what conditions. Uncontrolled message printing by libraries could potentially e.g. disturb application flow, expose implementation details, spam the user with irrelevant details, or break user interfaces.

The Logtalk message printing mechanism provides you with a set of predicates, defined in the `logtalk` built-in object, and some hook predicates. Two of the most important predicates are `print_message(Kind, Component, Term)`, which is used for printing a message, and `message_hook(Term, Kind, Component, Tokens)`, a user-defined hook predicate used for intercepting messages. The `Kind` argument is used to represent the nature of the message being printed. It can be e.g. a logging message, a warning message or an error message, In Logtalk this argument can be either an atom, e.g. `error`, or a compound term, e.g. `comment(loading)`. Using a compound term allows easy partitioning of messages of the same kind in different groups. The following kinds of message are recognized by default:

`banner`
: banner messages (used e.g. when loading tools or main application components; can be suppressed by setting the `report` flag to `warnings` or `off`)

`help`
: messages printed in reply for the user asking for help (mostly for helping port existing Prolog code)

`information` and `information(Group)`
: messages printed usually in reply to a user request for information

`silent` and `silent(Group)`
: not printed by default (but can be intercepted using the `message_hook/4` predicate)

`comment` and `comment(Group)`
: useful but usually not essential messages (can be suppressed by setting the `report` flag to `warnings` or `off`)

`warning` and `warning(Group)`
: warning messages (generated e.g. by the compiler; can be suppressed by turning off the `report` flag)

`error` and `error(Group)`
: error messages (generated e.g. by the compiler)

Note that you can define your own alternative message kind identifiers, for your own components, together with suitable definitions for their associated prefixes and output streams.

Messages are represented by atoms or compound terms, handy for machine-processing, and converted into a list of tokens, for human consumption. This conversion is performed using the multifile non-terminal `message_tokens(Term, Component)`. A simple example is:

```
:- multifile(logtalk::message_tokens//2).
:- dynamic(logtalk::message_tokens//2).

logtalk::message_tokens(loaded_settings_file(Path), core) -->
    ['Loaded settings file found on directory ~w'-[Path], nl, nl].
```

The `Component` argument is new in the Logtalk implementation and is useful to filter messages belonging to a specific component (e.g. the Logtalk compiler and runtime is identified by the atom `core`) and also to avoid conflicts when two components define the same message term (e.g. `banner`).

The following tokens can be used when translating a message:

`at_same_line`
: Signals a following part to a multi-part message with no line break in between; this token is ignored when it's not the first in the list of tokens

`flush`
: Flush the output stream (by calling the `flush_output/1` standard predicate)

`nl`
: Change line in the output stream

`Format-Arguments`
: `Format` must be an atom and `Arguments` must be a list of format arguments (the token arguments are passed to a call to the `format/3` de facto standard predicate)

`term(Term, Options)`
: `Term` can be any term and `Options` must be a list of valid `write_term/3` output options (the token arguments are passed to a call to the `write_term/3` standard predicate)

`ansi(Attributes, Format, Arguments)`
: Taken from SWI-Prolog; by default, do nothing; can be used for styled output

`begin(Kind, Var)`
: Taken from SWI-Prolog; by default, do nothing; can be used together with `end(Var)` to wrap a sequence of message tokens

`end(Var)`
: Taken from SWI-Prolog; by default, do nothing

There are also predicates for printing a list of tokens, `print_message_tokens(Stream, Prefix, Tokens)`, for hooking into printing an individual token, `print_message_token(Stream, Prefix, Token, Tokens)`, and for

setting default output stream and message prefixes, `message_prefix_stream(Kind, Component, Prefix, Stream)`. For example, the SWI-Prolog adapter file uses the `print_message_token/4` hook predicate to enable coloring of messages printed on a console.

Using the message printing mechanism in your applications and libraries is easy. Simply chose a unique component name for your application (e.g. the name of the application itself), call `logtalk::print_message/3` for every message that you may want to print, and define default translations for your message terms using the multifile non-terminal `logtalk::message_tokens/2`. For a full programming example, see e.g. the `lgtunit` tool source code.

Asking questions

Logtalk features a *structured question asking* mechanism that complements the message printing mechanism. This feature provides an abstraction for the common task of asking a user a question and reading back its reply. By default, this mechanism writes the question, writes a prompt, and reads the answer from the current user input and output streams but allows both steps to be intercepted, filtered, rewritten, and redirected. Two typical examples are using a GUI dialog for asking questions and automatically providing answers to specific questions.

The question asking mechanism works in tandem with the message printing mechanism, using it to print the question text and a prompt. It provides a asking predicate and a hook predicate, both declared and defined in the `logtalk` built-in object. The asking predicate, `ask_question(Kind, Component, Question, Check, Answer)`, is used for ask a question and read the answer. The hook predicate, `question_hook(Question, Kind, Component, Tokens, Check, Answer)`, is used for intercepting questions. The `Kind` argument is used to represent the nature of the question being asked. Its default value is `question` but it can be any atom or compound term, e.g. `question(parameters)`. Using a compound term allows easy partitioning of messages of the same kind in different groups. The `Check` argument is a closure that is converted into a checking goal taking as argument the user answer. The `ask_question/5` implements a read loop that terminates when this checking predicate is true. The question itself is a term that is translated into printing tokens using the `message_tokens/2` multifile predicate described above.

There is also a user-defined multifile predicate for setting default prompt and input streams, `question_prompt_stream(Kind, Component, Prompt, Stream)`.

An usage example of this mechanism can be found in the `debugger` tool where it's used to abstract the user interaction when tracing a goal execution in debug mode.

Predicate properties

We can find the properties of visible predicates by calling the `predicate_property/2` built-in method. For example:

```
| ?- bar::predicate_property(foo(_), Property).
```

Note that this method respects the predicate's scope declarations. For instance, the above call will only return properties for public predicates.

An object's set of visible predicates is the union of all the predicates declared for the object with all the built-in methods and all the Logtalk and Prolog built-in predicates.

The following predicate properties are supported:

`scope(Scope)`
: The predicate scope (useful for finding the predicate scope with a single call to `predicate_property/2`)

`public`, `protected`, `private`
: The predicate scope (useful for testing if a predicate have a specific scope)

`static`, `dynamic`
: All predicates are either static or dynamic (note, however, that a dynamic predicate can only be abolished if it was dynamically declared)

`logtalk`, `prolog`, `foreign`
: A predicate can be defined in Logtalk source code, Prolog code, or in foreign code (e.g. in C)

`built_in`
: The predicate is a built-in predicate

`multifile`
: The predicate is declared multifile (i.e. it can have clauses defined in several entities)

`meta_predicate(Template)`
: The predicate is declared as a meta-predicate with the specified template

`coinductive(Template)`
: The predicate is declared as a coinductive predicate with the specified template

`declared_in(Entity)`
: The predicate is declared (using a scope directive) in the specified entity

`defined_in(Entity)`
: The predicate definition is looked up in the specified entity (note that this property does not necessarily imply that clauses for the predicate exist in `Entity`; the predicate can simply be false as per the closed-world assumption)

`redefined_from(Entity)`
: The predicate is a redefinition of a predicate definition inherited from the specified entity

`non_terminal(NonTerminal//Arity)`
: The predicate resulted from the compilation of the specified grammar rule non-terminal

`alias_of(Predicate)`
: The predicate (name) is an alias for the specified predicate

`alias_declared_in(Entity)`
: The predicate alias is declared in the specified entity

`synchronized`
: The predicate is declared as synchronized (i.e. it's a deterministic predicate synchronized using a mutex when using a backend Prolog compiler supporting a compatible multi-threading implementation)

Some properties are only available when the entities are defined in source files and when those source files are compiled with the `source_data` flag turned on:

`inline`
: The predicate definition is inlined

`auxiliary`
: The predicate is not user-defined but rather automatically generated by the compiler or the term-expansion mechanism

`mode(Mode, Solutions)`
: Instantiation, type, and determinism mode for the predicate (which can have multiple modes)

`info(ListOfPairs)`
: Documentation key-value pairs as specified in the user-defined `info/2` directive

`number_of_clauses(N)`
: The number of clauses for the predicate existing at compilation time (note that this property is not updated at runtime when asserting and retracting clauses for dynamic predicates)

`number_of_rules(N)`
: The number of rules for the predicate existing at compilation time (note that this property is not updated at runtime when asserting and retracting clauses for dynamic predicates)

`declared_in(Entity, Line)`
: The predicate is declared (using a scope directive) in the specified entity in a source file at the specified line (if applicable)

`defined_in(Entity, Line)`
: The predicate is defined in the specified entity in a source file at the specified line (if applicable)

`redefined_from(Entity, Line)`
: The predicate is a redefinition of a predicate definition inherited from the specified entity, which is defined in a source file at the specified line (if applicable)

`alias_declared_in(Entity, Line)`
: The predicate alias is declared in the specified entity in a source file at the specified line (if applicable)

The properties `declared_in/1-2`, `defined_in/1-2`, and `redefined_from/1-2` do not apply to built-in methods and Logtalk or Prolog built-in predicates. Note that if a predicate is declared in a category imported by the object, it will be the category name — not the object name — that will be returned by the property `declared_in/1`. The same is true for protocol declared predicates.

Finding declared predicates

We can find, by backtracking, all visible user predicates by calling the `current_predicate/1` built-in method. This method respects the predicate's scope declarations. For instance, the following call:

```
| ?- some_object::current_predicate(Name/Arity).
```

will only return user predicates that are declared public. The predicate property `non_terminal/1` may be used to retrieve all grammar rule non-terminals declared for an object. For example:

```
current_non_terminal(Object, NonTerminal//Args) :-
    Object::current_predicate(Name/Arity),
    functor(Predicate, Functor, Arity),
    Object::predicate_property(Predicate, non_terminal(NonTerminal//Args)).
```

Usually, the non-terminal and the corresponding predicate share the same functor but users should not rely on this always being true.

Calling Prolog built-in predicates

In predicate definitions, predicate calls which are not prefixed with a message sending operator (either `::` or `^^`), are compiled to either calls to local predicates or as calls to Logtalk/Prolog built-in predicates. A predicate call is compiled as a call to a local predicate if the object (or category) contains a scope directive, a definition for the called predicate, or a dynamic declaration for it. When the object (or category) does not contain either a definition of the called predicate or a corresponding dynamic declaration, Logtalk tests if the call corresponds to a Logtalk or Prolog built-in predicate. Calling a predicate which is neither a local predicate nor a Logtalk/Prolog built-in predicate results in a compile time warning. This means that, in the following example:

```
foo :-
    ...,
    write(bar),
    ...
```

the call to the predicate `write/1` will be compiled as a call to the corresponding Prolog built-in predicate unless the object (or category) encapsulating the above definition also contains a predicate named `write/1` or a dynamic declaration for the predicate.

When calling non-standard Prolog built-in predicates or using non-standard Prolog arithmetic functions, you may run into portability problems while trying your applications with different back-end Prolog compilers (non-standard predicates and non-standard arithmetic functions are often specific to a Prolog compiler). You may use the Logtalk compiler flag `portability/1` to help check for problematic calls in your code.

Calling Prolog non-standard meta-predicates

Prolog built-in meta-predicates may only be called locally within objects or categories, i.e. they cannot be used as messages. Compiling calls to non-standard, Prolog built-in meta-predicates can be tricky, however, as there is no standard way of checking if a built-in predicate is also a meta-predicate and finding out which are its meta-arguments. But Logtalk supports override the original meta-predicate template if not programmatically available or usable. For example, assume a `det_call/1` Prolog built-in meta-predicate that takes a goal as argument. We can add to the object (or category) calling it the directive:

```
:- meta_predicate(user:det_call(0)).
```

Another solution is to explicitly declare all non-standard Prolog meta-predicates in the corresponding adapter file using the internal predicate `'$lgt_prolog_meta_predicate'/3`. For example:

```
'$lgt_prolog_meta_predicate'(det_call(_), det_call(0), predicate).
```

The third argument can be either the atom `predicate` or the atom `control_construct`, a distinction that is useful when compiling in debug mode.

Calling Prolog user-defined predicates

Prolog user-defined predicates can be called from within objects or categories by using the `{}/1` compiler bypass control construct. For example:

```
foo :-
    ...,
    {bar},
    ...
```

In alternative, you can also use the `uses/2` directive and write:

```
:- uses(user, [bar/0]).

foo :-
    ...,
    bar,
    ...
```

Note that `user` is a pseudo-object in Logtalk containing all predicate definitions that are not encapsulated (either in a Logtalk entity or a Prolog module).

Calling Prolog module predicates

To call Prolog module predicates from within objects or categories you can use simply write:

```
foo :-
    ...,
    module:bar,
    ...
```

You can also use in alternative the `use_module/2` directive:

```
:- use_module(module, [bar/0]).

foo :-
    ...,
    bar,
    ...
```

Note that the first argument of the `use_module/2`, when used within an object or a category, is a module name, not a file name. The actual module code should be loaded prior to compilation of Logtalk that uses it. In particular, programmers should not expect that the module be auto-loaded (when using back-end Prolog compilers supporting an autoloading mechanism).

When the module predicate is a meta-predicate but for some reason you don't want its calls to be compiled as such, you can use the `{}/1` compiler bypass control construct as before:

```
foo :-
    ...,
    {module:bar},
    ...
```

This workaround is sometimes necessary when calling module meta-predicates whose meta-predicate templates are ambiguous and cannot be processed by the Logtalk compiler (note, however, that it's often possible to specify an overriding meta-predicate directive within the object or category making the call as explained above).

Inheritance

The inheritance mechanisms found on object-oriented programming languages allow us the specialization of previously defined objects, avoiding the unnecessary repetition of code. In the context of logic programming, we can interpret inheritance as a form of theory extension: an object will virtually contain, besides its own predicates, all the predicates inherited from other objects that are not redefined by itself.

Logtalk uses a depth-first lookup procedure for finding predicate declarations and predicate definitions, as explained below. The lookup procedures locate the entities holding the predicate declaration and the predicate definition using, respectively, the predicate indicator and the predicate template (constructed from the predicate indicator).

The `alias/2` predicate directive may be used to defining alternative names for inherited predicates for solving inheritance conflicts and for giving access to all inherited definitions.

Protocol inheritance

Protocol inheritance refers to the inheritance of predicate declarations (scope directives). These can be contained in objects, in protocols, or in categories. Logtalk supports single and multi-inheritance of protocols: an object or a category may implement several protocols and a protocol may extend several protocols.

Search order for prototype hierarchies

The search order for predicate declarations is first the object, second the implemented protocols (and the protocols that these may extend), third the imported categories (and the protocols that they may implement), and last the objects that the object extends. This search is performed in a depth-first way. When an object inherits two different declarations for the same predicate, by default, only the first one will be considered.

Search order for class hierarchies

The search order for predicate declarations starts in the object classes. Following the classes declaration order, the search starts in the classes implemented protocols (and the protocols that these may extend), third the classes imported categories (and the protocols that they may implement), and last the superclasses of the object classes. This search is performed in a depth-first way. If the object inherits two different declarations for the same predicate, by default only the first one will be considered.

Implementation inheritance

Implementation inheritance refers to the inheritance of predicate definitions. These can be contained in objects or in categories. Logtalk supports multi-inheritance of implementation: an object may import several categories or extend, specialize, or instantiate several objects.

Search order for prototype hierarchies

The search order for predicate definitions is similar to the search for predicate declarations except that implemented protocols are ignored (as they can only contain predicate directives).

Search order for class hierarchies

The search order for predicate definitions is similar to the search for predicate declarations except that implemented protocols are ignored (as they can only contain predicate directives) and that the search starts at the instance itself (that received the message) before proceeding, if no predicate definition is found there, to the instance classes.

Inheritance versus predicate redefinition

When we define a predicate that is already inherited from other object, the inherited definitions are hidden by the new definitions. This is called overriding inheritance: a local definition overrides any inherited ones. For example, assume that we have the following two objects:

```
:- object(root).

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

    bar(root).

    foo(root).

:- end_object.

:- object(descendant,
    extends(root)).

    foo(descendant).

:- end_object.
```

After compiling and loading these objects, we can check the overriding behavior by trying the following queries:

```
| ?- root::(bar(Bar), foo(Foo)).

Bar = root
Foo = root
yes

| ?- descendant::(bar(Bar), foo(Foo)).

Bar = root
Foo = descendant
yes
```

However, we can explicitly program other behaviors. Let us see a few examples.

Specialization inheritance

Specialization of inherited definitions: the new definition uses the inherited definitions, adding to this new code. This is accomplished by calling the `^^/1` operator in the new definition.

```
:- object(root).

    :- public(init/0).

    init :-
        write('root init'), nl.

:- end_object.

:- object(descendant,
    extends(root)).

    init :-
        write('descendant init'), nl,
        ^^init.

:- end_object.

| ?- descendant::init.

descendant init
root init

yes
```

Union inheritance

Union of the new with the inherited definitions: all the definitions are taken into account, the calling order being defined by the inheritance mechanisms. This can be accomplished by writing a clause that just calls, using the `^^/1` operator, the

inherited definitions. The relative position of this clause among the other definition clauses sets the calling order for the local and inherited definitions.

```
:- object(root).

    :- public(foo/1).

    foo(1).
    foo(2).

:- end_object.

:- object(descendant,
    extends(root)).

    foo(3).
    foo(Foo) :-
        ^^foo(Foo).

:- end_object.

| ?- descendant::foo(Foo).

Foo = 3 ;
Foo = 1 ;
Foo = 2 ;

no
```

Selective inheritance

Hiding some of the inherited definitions, or differential inheritance: this form of inheritance is normally used in the representation of exceptions to generic definitions. Here we will need to use the `^^/1` operator to test and possibly reject some of the inherited definitions.

```
:- object(bird).

    :- public(mode/1).

    mode(walks).
    mode(flies).

:- end_object.

:- object(penguin,
    extends(bird)).

    mode(swims).
    mode(Mode) :-
        ^^mode(Mode),
        Mode \= flies.

:- end_object.

| ?- penguin::mode(Mode).

Mode = swims ;
Mode = walks ;

no
```

Public, protected, and private inheritance

To make all public predicates declared via implemented protocols, imported categories, or inherited objects protected or to make all public and protected predicates private we prefix the entity's name with the corresponding keyword. For instance:

```
:- object(Object,
    implements(private::Protocol)).       % all the Protocol public and protected
    ...                                   % predicates become private predicates
:- end_object.                            % for the Object clients
```

or:

```
:- object(Class,
    specializes(protected::Superclass)).  % all the Superclass public predicates become
    ...                                   % protected predicates for the Class clients
:- end_object.
```

Omitting the scope keyword is equivalent to using the public scope keyword. For example:

```
:- object(Object,
    imports(public::Category)).
    ...
:- end_object.
```

This is the same as:

```
:- object(Object,
    imports(Category)).
    ...
:- end_object.
```

This way we ensure backward compatibility with older Logtalk versions and a simplified syntax when protected or private inheritance are not used.

Composition versus multiple inheritance

It is not possible to discuss inheritance mechanisms without referring to the long and probably endless debate on single versus multiple inheritance. The single inheritance mechanism can be implemented in an very efficient way, but it imposes several limitations on reusing, even if the multiple characteristics we intend to inherit are orthogonal. On the other hand, the multiple inheritance mechanisms are attractive in their apparent capability of modeling complex situations. However, they include a potential for conflict between inherited definitions whose variety does not allow a single and satisfactory solution for all the cases.

Until now, no solution that we might consider satisfactory for all the problems presented by the multiple inheritance mechanisms has been found. From the simplicity of some extensions that use the Prolog search strategy like [McCabe 92] or [Moss 94] and to the sophisticated algorithms of CLOS [Bobrow 88], there is no adequate solution for all the situations. Besides, the use of multiple inheritance carries some complex problems in the domain of software engineering, particularly in the reuse and maintenance of the applications. All these problems are substantially reduced if we preferably use in our software development composition mechanisms instead of specialization mechanisms [Taenzer 89]. Multiple inheritance can and should be seen more as a useful analysis and project abstraction, than as an implementation technique [Shan 93]. Logtalk provides first-class support for software composition using *categories*.

Nevertheless, Logtalk supports multi-inheritance by enabling an object to extend, instantiate, or specialize more than one object. The current Logtalk release provides a predicate directive, `alias/2`, which may be used to solve some multi-inheritance conflicts. Lastly, it should be noted that the multi-inheritance support does not compromise performance when we use single-inheritance.

Event-driven programming

The addition of event-driven programming capacities to the Logtalk language [Moura 94] is based on a simple but powerful idea:

> The computations must result, not only from message sending, but also from the **observation** of message sending.

The need to associate computations to the occurrence of events was very early recognized in several knowledge representation languages, in some programming languages [Stefik 86, Moon 86], and in the implementation of operative systems [Tanenbaum 87] and graphical user interfaces.

With the integration between object-oriented and event-driven programming, we intend to achieve the following goals:

- Minimize the coupling between objects. An object should only contain what is intrinsic to it. If an object observes another object, that means that it should depend only on the (public) protocol of the object observed, and not on the implementation of that same protocol.

- Provide a mechanism for building reflexive systems in Logtalk based on the dynamic behavior of objects in complement to the reflective information of the object's contents and relations.

- Provide a mechanism for easily defining method pre- and post-conditions that can be toggled using the `events` compiler flag. The pre- and post-conditions may be defined in the same object containing the methods or distributed between several objects acting as method monitors.

Definitions

The words *event* and *monitor* have multiple meanings in computer science, so, to avoid misunderstandings, it is advisable that we start by defining them in the Logtalk context.

Event

In an object-oriented system, all computations start through message sending. It thus becomes quite natural to declare that the only event that can occur in this kind of system is precisely the sending of a message. An event can thus be represented by the ordered tuple `(Object, Message, Sender)`.

If we consider message processing an indivisible activity, we can interpret the sending of a message and the return of the control to the object that has sent the message as two distinct events. This distinction allows us to have a more precise control over a system dynamics. In Logtalk, these two types of events have been named `before` and `after`, respectively

for message sending and returning. Therefore, we end up by representing an event by the ordered tuple `(Event, Object, Message, Sender)`.

The implementation of the event notion in Logtalk enjoys the following properties:

Independence between the two types of events
: We can choose to watch only one event type or to process each one of the events associated to a message sending in an independent way.

All events are automatically generated by the message sending mechanism
: The task of generating events is accomplished, in a transparent way, by the message sending mechanism. The user just defines which are the events in which he is interested in.

The events watched at any moment can be dynamically changed during program execution
: The notion of event allows the user not only to have the possibility of observing, but also of controlling and modifying an application behavior, namely by dynamically changing the observed events during program execution. It is our goal to provide the user with the possibility of modeling the largest possible number of situations.

Monitor

Complementary to the notion of event is the notion of monitor. A monitor is an object that is automatically notified by the message sending mechanisms whenever certain events occur. A monitor should naturally define the actions to be carried out whenever a monitored event occurs.

The implementation of the monitor notion in Logtalk enjoys the following properties:

Any object can act as a monitor
: The monitor status is a role that any object can perform during its existence. The minimum protocol necessary is declared in the built-in protocol `monitoring`. Strictly speaking, the reference to this protocol is only needed when specializing event handlers. Nevertheless, it is considered good programming practice to always refer the protocol when defining event handlers.

Unlimited number of monitors for each event
: Several monitors can observe the same event because of distinct reasons. Therefore, the number of monitors per event is bounded only by the available computing resources.

The monitor status of an object can be dynamically changed in runtime
: This property does not imply that an object must be dynamic to act as a monitor (the monitor status of an object is not stored in the object).

The execution of actions, defined in a monitor, associated to each event, never affects the term that denotes the message involved
: In other words, if the message contains uninstantiated variables, these are not affected by the acting of monitors associated to the event.

Event generation

For each message that is sent (using the `::/2` message sending mechanism) the runtime system automatically generates two events. The first — `before event` — is generated when the message is sent. The second — `after event` — is generated after the message has successfully been executed.

Communicating events to monitors

Whenever a spied event occurs, the message sending mechanisms call the corresponding event handlers directly for all registered monitors. These calls are made bypassing the message sending primitives in order to avoid potential endless

loops. The event handlers consist in user definitions for the public predicates declared in the `monitoring` built-in protocol (one for each event kind; see below for more details).

Performance concerns

Ideally, the existence of monitored messages should not affect the processing of the remaining messages. On the other hand, for each message that has been sent, the system must verify if its respective event is monitored. Whenever possible, this verification should be performed in constant time and independently from the number of monitored events. The events representation takes advantage of the first argument indexing performed by most Prolog compilers, which ensure — in the general case — an access in constant time.

Event-support can be turned off on a per-object (or per-category) basis using the compiler flag `events/1`. With event-support turned off, Logtalk uses optimized code for processing message sending calls that skips the checking of monitored events, resulting in a small but measurable performance improvement.

Monitor semantics

The established semantics for monitors actions consists on considering its success as a necessary condition so that a message can succeed:

- All actions associated to events of type `before` must succeed, so that the message processing can start.

- All actions associated to events of type `after` also have to succeed so that the message itself succeeds. The failure of any action associated to an event of type `after` forces backtracking over the message execution (the failure of a monitor never causes backtracking over the preceding monitor actions).

Note that this is the most general choice. If we wish a transparent presence of monitors in a message processing, we just have to define the monitor actions in such a way that they never fail (which is very simple to accomplish).

Activation order of monitors

Ideally, whenever there are several monitors defined for the same event, the calling order should not interfere with the result. However, this is not always possible. In the case of an event of type `before`, the failure of a monitor prevents a message from being sent and prevents the execution of the remaining monitors. In case of an event of type `after`, a monitor failure will force backtracking over message execution. Different orders of monitor activation can therefore lead to different results if the monitor actions imply object modifications unrecoverable in case of backtracking. Therefore, the order for monitor activation should be assumed as arbitrary. In effect, to assume or to try to impose a specific sequence requires a global knowledge of an application dynamics, which is not always possible. Furthermore, that knowledge can reveal itself as incorrect if there is any changing in the execution conditions. Note that, given the independence between monitors, it does not make sense that a failure forces backtracking over the actions previously executed.

Event handling

Logtalk provides three built-in predicates for event handling. These predicates enable you to find what events are defined, to define new events and to abolish events when they are no longer needed. If you plan to use events extensively in your application, then you should probably define a set of objects that use the built-in predicates described below to implement more sophisticated and high-level behavior.

Defining new events

New events can be defined using the Logtalk built-in predicate `define_events/5`:

```
| ?- define_events(Event, Object, Message, Sender, Monitor).
```

Note that if any of the `Event`, `Object`, `Message`, and `Sender` arguments is a free variable or contains free variables, this call will define a **set** of matching events.

Abolishing defined events

Events that are no longer needed may be abolished using the `abolish_events/5` built-in predicate:

```
| ?- abolish_events(Event, Object, Message, Sender, Monitor).
```

If called with free variables, this goal will remove all matching events.

Finding defined events

The events that are currently defined can be retrieved using the Logtalk built-in predicate `current_event/5`:

```
| ?- current_event(Event, Object, Message, Sender, Monitor).
```

Note that this predicate will return **sets** of matching events if some of the returned arguments are free variables or contain free variables.

Defining event handlers

The `monitoring` built-in protocol declares two public predicates, `before/3` and `after/3`, that are automatically called to handle `before` and `after` events. Any object that plays the role of monitor must define one or both of these event handler methods:

```
before(Object, Message, Sender) :-
    ... .

after(Object, Message, Sender) :-
    ... .
```

The arguments in both methods are instantiated by the message sending mechanisms when a monitored event occurs. For example, assume that we want to define a monitor called `tracer` that will track any message sent to an object by printing a describing text to the standard output. Its definition could be something like:

```
:- object(tracer,
    implements(monitoring)).    % built-in protocol for event handler methods

    before(Object, Message, Sender) :-
        write('call: '), writeq(Object), write(' <-- '), writeq(Message),
        write(' from '), writeq(Sender), nl.

    after(Object, Message, Sender) :-
        write('exit: '), writeq(Object), write(' <-- '), writeq(Message),
        write(' from '), writeq(Sender), nl.

:- end_object.
```

Assume that we also have the following object:

```
:- object(any).

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

    bar(bar).

    foo(foo).

:- end_object.
```

After compiling and loading both objects (note that the object `any` must be compiled with the flag `events(allow)`), we can start tracing every message sent to any object by calling the `define_events/5` built-in predicate:

```
| ?- define_events(_, _, _, _, tracer).

yes
```

From now on, every message sent to any object will be traced to the standard output stream:

```
| ?- any::bar(X).

call: any <-- bar(X) from user
exit: any <-- bar(bar) from user
X = bar

yes
```

To stop tracing, we can use the `abolish_events/5` built-in predicate:

```
| ?- abolish_events(_, _, _, _, tracer).

yes
```

The `monitoring` protocol declares the event handlers as public predicates. If necessary, protected or private implementation of the protocol may be used in order to change the scope of the event handler predicates. Note that the message sending processing mechanisms are able to call the event handlers irrespective of their scope. Nevertheless, the scope of the event handlers may be restricted in order to prevent other objects from calling them.

Multi-threading programming

Logtalk provides **experimental** support for multi-threading programming on selected Prolog compilers. Logtalk makes use of the low-level Prolog built-in predicates that implement message queues and interface with POSIX threads and mutexes (or a suitable emulation), providing a small set of high-level predicates and directives that allows programmers to easily take advantage of modern multi-processor and multi-core computers without worrying about the details of creating, synchronizing, or communicating with threads. Logtalk multi-threading programming integrates with object-oriented programming providing a threaded engines API, enabling objects and categories to prove goals concurrently, and supporting synchronous and asynchronous messages.

Enabling multi-threading support

Multi-threading support may be disabled by default. It can be enabled on the Prolog adapter files of supported compilers by setting the read-only compiler flag `threads` to `supported`.

Enabling objects to make multi-threading calls

The `threaded/0` object directive is used to enable an object to make multi-threading calls:

```
:- threaded.
```

This directive results in the automatic creation and set up of an object message queue when the object is loaded or created at runtime. Object message queues are used for exchanging thread notifications and for storing concurrent goal solutions and replies to the *multi-threading calls* made within the object. The message queue for the pseudo-object `user` is automatically created at Logtalk startup (provided that multi-threading programming is supported and enabled for the chosen Prolog compiler).

Multi-threading built-in predicates

Logtalk provides a small set of built-in predicates for multi-threading programming. For simple tasks where you simply want to prove a set of goals, each one in its own thread, Logtalk provides a `threaded/1` built-in predicate. The remaining predicates allow for fine-grained control, including postponing retrieving of thread goal results at a later time, supporting non-deterministic thread goals, and making *one-way* asynchronous calls. Together, these predicates provide high-level support for multi-threading programming, covering most common use cases.

Proving goals concurrently using threads

A set of goals may be proved concurrently by calling the Logtalk built-in predicate `threaded/1`. Each goal in the set runs in its own thread.

When the `threaded/1` predicate argument is a *conjunction* of goals, the predicate call is akin to *and-parallelism*. For example, assume that we want to find all the prime numbers in a given interval, `[N, M]`. We can split the interval in two parts and then span two threads to compute the primes numbers in each sub-interval:

```
prime_numbers(N, M, Primes) :-
    M > N,
    N1 is N + (M - N) // 2,
    N2 is N1 + 1,
    threaded((
        prime_numbers(N2, M, [], Acc),
        prime_numbers(N, N1, Acc, Primes)
    )).

prime_numbers(N, M, Acc, Primes) :-
    ...
```

The `threaded/1` call terminates when the two implicit threads terminate. In a computer with two or more processors (or with a processor with two or more cores) the code above can be expected to provide better computation times when compared with single-threaded code for sufficiently large intervals.

When the `threaded/1` predicate argument is a *disjunction* of goals, the predicate call is akin to *or-parallelism*, here reinterpreted as a set of goals *competing* to find a solution. For example, consider the different methods that we can use to find the roots of real functions. Depending on the function, some methods will faster than others. Some methods will converge into the solution while others may diverge and never find it. We can try all the methods simultaneously by writing:

```
find_root(Function, A, B, Error, Zero, Algorithm) :-
    threaded((
            (bisection::find_root(Function, A, B, Error, Zero), Algorithm = bisection)
        ;   (newton::find_root(Function, A, B, Error, Zero), Algorithm = newton)
        ;   (muller::find_root(Function, A, B, Error, Zero), Algorithm = muller)
    )).
```

The above `threaded/1` goal succeeds when one of the implicit threads succeeds in finding the function root, leading to the termination of all the remaining competing threads.

The `threaded/1` built-in predicate is most useful for lengthy, independent deterministic computations where the computational costs of each goal outweigh the overhead of the implicit thread creation and management.

Proving goals asynchronously using threads

A goal may be proved asynchronously using a new thread by calling the Logtalk built-in predicate `threaded_call/1`. Calls to this predicate are always true and return immediately (assuming a callable argument). The term representing the goal is copied, not shared with the thread. The thread computes the first solution to the goal, posts it to the message queue of the object from where the `threaded_call/1` predicate was called, and suspends waiting for either a request for an alternative solution or for the program to commit to the current solution.

The results of proving a goal asynchronously in a new thread may be later retrieved by calling the Logtalk built-in predicate `threaded_exit/1` within the same object where the call to the `threaded_call/1` predicate was made. The `threaded_exit/1` calls suspend execution until the results of the `threaded_call/1` calls are sent back to the object message queue.

The `threaded_exit/1` predicate allow us to retrieve alternative solutions through backtracking (if you want to commit to the first solution, you may use the `threaded_once/1` predicate instead of the `threaded_call/1` predicate). For example, assuming a `lists` object implementing the usual `member/2` predicate, we could write:

```
| ?- threaded_call(lists::member(X, [1,2,3])).

X = _G189
yes

| ?- threaded_exit(lists::member(X, [1,2,3])).

X = 1 ;
X = 2 ;
X = 3 ;
no
```

In this case, the `threaded_call/1` and the `threaded_exit/1` calls are made within the pseudo-object *user*. The implicit thread running the `lists::member/2` goal suspends itself after providing a solution, waiting for a request to an alternative solution; the thread is automatically terminated when the runtime engine detects that backtracking to the `threaded_exit/1` call is no longer possible.

Calls to the `threaded_exit/1` predicate block the caller until the object message queue receives the reply to the asynchronous call. The predicate `threaded_peek/1` may be used to check if a reply is already available without removing it from the thread queue. The `threaded_peek/1` predicate call succeeds or fails immediately without blocking the caller. However, keep in mind that repeated use of this predicate is equivalent to polling a message queue, which may severely hurt performance.

Be careful when using the `threaded_exit/1` predicate inside failure-driven loops. When all the solutions have been found (and the thread generating them is therefore terminated), re-calling the predicate will generate an exception. Note that failing instead of throwing an exception is not an acceptable solution as it could be misinterpreted as a failure of the `threaded_exit/1` argument.

The example on the previous section with prime numbers could be rewritten using the `threaded_call/1` and `threaded_exit/1` predicates:

```
prime_numbers(N, M, Primes) :-
    M > N,
    N1 is N + (M - N) // 2,
    N2 is N1 + 1,
    threaded_call(prime_numbers(N2, M, [], Acc)),
    threaded_call(prime_numbers(N, N1, Acc, Primes)),
    threaded_exit(prime_numbers(N2, M, [], Acc)),
    threaded_exit(prime_numbers(N, N1, Acc, Primes)).

prime_numbers(N, M, Acc, Primes) :-
    ...
```

When using asynchronous calls, the link between a `threaded_exit/1` call and the corresponding `threaded_call/1` call is established using unification. If there are multiple `threaded_call/1` calls for a matching `threaded_exit/1`

call, the connection can potentially be established with any of them. Nevertheless, you can easily use a tag the calls by using the extended `threaded_call/2` and `threaded_exit/2` built-in predicates. For example:

```
?- threaded_call(member(X, [1,2,3]), Tag).

Tag = 1
yes

?- threaded_call(member(X, [1,2,3]), Tag).

Tag = 2
yes

?- threaded_exit(member(X, [1,2,3]), 2).

X = 1 ;
X = 2 ;
X = 3
yes
```

When using these predicates, the tags shall be considered as an opaque term; users shall not rely on its type.

One-way asynchronous calls

Sometimes we want to prove a goal in a new thread without caring about the results. This may be accomplished by using the built-in predicate `threaded_ignore/1`. For example, assume that we are developing a multi-agent application where an agent may send an "happy birthday" message to another agent. We could write:

```
..., threaded_ignore(agent::happy_birthday), ...
```

The call succeeds with no reply of the goal success, failure, or even exception ever being sent back to the object making the call. Note that this predicate implicitly performs a deterministic call of its argument.

Asynchronous calls and synchronized predicates

Proving a goal asynchronously using a new thread may lead to problems when the goal results in side-effects such as input/output operations or modifications to an object database. For example, if a new thread is started with the same goal before the first one finished its job, we may end up with mixed output, a corrupted database, or unexpected goal failures. In order to solve this problem, predicates (and grammar rule non-terminals) with side-effects can be declared as *synchronized* by using the `synchronized/1` predicate directive. Proving a query to a synchronized predicate (or synchronized non-terminal) is internally protected by a mutex, thus allowing for easy thread synchronization. For example:

```
:- synchronized(db_update/1).   % ensure thread synchronization

db_update(Update) :-            % predicate with side-effects
    ...
```

A second example: assume an object defining two predicates for writing, respectively, even and odd numbers in a given interval to the standard output. Given a large interval, a goal such as:

```
| ?- threaded_call(obj::odd_numbers(1,100)), threaded_call(obj::even_numbers(1,100)).

1 3 2 4 6 8 5 7 10 ...
...
```

will most likely result in a mixed up output. By declaring the `odd_numbers/2` and `even_numbers/2` predicates synchronized:

```
:- synchronized([
    odd_numbers/2,
    even_numbers/2]).
```

one goal will only start after the other one finished:

```
| ?- threaded_ignore(obj::odd_numbers(1,99)), threaded_ignore(obj::even_numbers(1,99)).

1 3 5 7 9 11 ...
...
2 4 6 8 10 12 ...
...
```

Note that, in a more realistic scenario, the two `threaded_ignore/1` calls would be made concurrently from different objects. Using the same synchronized directive for a set of predicates imply that they all use the same mutex, as required for this example.

As each Logtalk entity is independently compiled, this directive must be included in every object or category that contains a definition for the described predicate, even if the predicate declaration is inherited from another entity, in order to ensure proper compilation. Note that a synchronized predicate cannot be declared dynamic. To ensure atomic updates of a dynamic predicate, declare as synchronized the predicate performing the update.

Synchronized predicates may be used as wrappers to messages sent to objects that are not multi-threading aware. For example, assume a `random` object defining a `random/1` predicate that generates random numbers, using side-effects on its implementation (e.g. for storing the generator seed). We can specify and define e.g. a `sync_random/1` predicate as follows:

```
:- synchronized(sync_random/1).

sync_random(Random) :-
    random::random(Random).
```

and then always use the `sync_random/1` predicate instead of the predicate `random/1` from multi-threaded code.

The synchronization entity and predicate directives may be used when defining objects that may be reused in both single-threaded and multi-threaded Logtalk applications. The directives are simply ignored (i.e. the synchronized predicates are interpreted as normal predicates) when the objects are used in a single-threaded application.

Synchronizing threads through notifications

Declaring a set of predicates as synchronized can only ensure that they are not executed at the same time by different threads. Sometimes we need to suspend a thread not on a synchronization lock but on some condition that must hold true for a thread goal to proceed. I.e. we want a thread goal to be suspended until a condition becomes true instead of simply failing. The built-in predicate `threaded_wait/1` allows us to suspend a predicate execution (running in its own thread) until a notification is received. Notifications are posted using the built-in predicate `threaded_notify/1`. A notification is a Prolog term that a programmer chooses to represent some condition becoming true. Any Prolog term can be used as a notification argument for these predicates. Related calls to the `threaded_wait/1` and `threaded_notify/1` must be made within the same object, *this*, as the object message queue is used internally for posting and retrieving notifications.

Each notification posted by a call to the `threaded_notify/1` predicate is consumed by a single `threaded_wait/1` predicate call (i.e. these predicates implement a peer-to-peer mechanism). Care should be taken to avoid deadlocks when two (or more) threads both wait and post notifications to each other.

Engines

Threaded *engines* provide an alternative to the multi-threading predicates described in the previous sections. An engine is a computing thread whose solutions can be lazily computed and retrieved. In addition, an engine also supports a term queue that allows passing arbitrary terms to the engine.

An engine is created by calling the `threaded_engine_create/3` built-in predicates. For example:

```
| ?- threaded_engine_create(X, member(X, [1,2,3]), worker).
yes
```

The first argument is an *answer template* to be used for retrieving solution bindings. The user can name the engine, as in this example where the atom `worker` is used, or have the runtime generate a name, which should be treated as an opaque term.

Engines are scoped by the object within which the `threaded_engine_create/3` call takes place. Thus, different objects can create engines with the same names with no conflicts. Morevover, engines share the visible predicates of the object creating them.

The engine computes the first solution of its goal argument and suspends waiting for it to be retrieved. Solutions can be retrieved one at a time using the `threaded_engine_next/2` built-in predicate:

```
| ?- threaded_engine_next(worker, X).
X = 1
yes
```

The call blocks until a solution is available and fails if there are no solutions left. After returning a solution, this predicate signals the engine to start computing the next one. Note that this predicate is deterministic. In constrast with the `threaded_exit/1-2` built-in predicates, retrieving the next solution requires calling the predicate again instead of by backtracking into its call.

There is also an alternative reified version of the predicate, `threaded_engine_next_reified/2`, which returns `the(Answer)`, `no`, and `exception(Error)` terms as answers.

Engines must be explicitly terminated using the `threaded_engine_destroy/1` built-in predicate:

```
| ?- threaded_engine_destroy(worker).
yes
```

A common usage pattern for engines is to define a recursive predicate that uses the engine term queue to retrieve a task to be performed. For example, assume we define the following predicate:

```
loop :-
 threaded_engine_fetch(Task),
 handle(Task),
 loop.
```

The `threaded_engine_fetch/1` built-in predicate fetches a task for the engine term queue. The engine clients would use the `threaded_engine_post/2` built-in predicate to post tasks into the engine term queue. The engine would be created using the call:

```
| ?- threaded_engine_create(none, loop, worker).
yes
```

The `handle/1` predicate, after performing a task, can use the `threaded_engine_yield/1` built-in predicate to make the task results available for consumption using the `threaded_engine_next/2` built-in predicate. Blocking semantics are used by these two predicates: the `threaded_engine_yield/1` predicate blocks until the returned solution is consumed while the `threaded_engine_next/2` predicate blocks until a solution becomes available.

Multi-threading performance

The performance of multi-threading applications is highly dependent on the back-end Prolog compiler, on the operating-system, and on the use of dynamic binding and dynamic predicates. All compatible back-end Prolog compilers that support multi-threading features make use of POSIX threads or *pthreads*. The performance of the underlying pthreads implementation can exhibit significant differences between operating systems. An important point is synchronized access to dynamic predicates. As different threads may try to simultaneously access and update dynamic predicates, these operations must be protected by a lock, usually implemented using a mutex. Poor mutex lock operating-system performance, combined with a large number of collisions by several threads trying to acquire the same lock, often result in severe performance penalties. Thus, whenever possible, avoid using dynamic predicates and dynamic binding.

Error handling

All error handling is done in Logtalk by using the ISO defined `catch/3` and `throw/1` predicates [ISO 95]. Errors thrown by Logtalk have the following format:

```
error(Error, logtalk(Goal, ExecutionContext))
```

In this exception term, `Goal` is the goal that triggered the error `Error` and `ExecutionContext` is the context in which `Goal` is called. For example:

```
error(permission_error(modify,private_predicate,p), logtalk(foo::abolish(p/0),_))
```

Note, however, that `Goal` and `ExecutionContext` can be variables when the corresponding information is not available (e.g. due to compiler optimizations that throw away the necessary error context information). The `ExecutionContext` argument is an opaque term that can be decoded using the `logtalk::execution_context/7` predicate.

Compiler warnings and errors

The current Logtalk compiler uses the `read_term/3` ISO Prolog defined built-in predicate to read and compile a Logtalk source file. One consequence of this is that invalid Prolog terms or syntax errors may abort the compilation process with limited information given to the user (due to the inherent limitations of the `read_term/3` predicate).

If all the terms in a source file are valid, then there is a set of errors or potential errors, described below, that the compiler will try to detect and report, depending on the used compiler flags (see the Writing, running, and debugging programs section of this manual for details).

Unknown entities

The Logtalk compiler warns about any referenced entity that is not currently loaded. The warning may reveal a misspell entity name or just an entity that it will be loaded later. Out-of-oder loading should be avoided when possible as it prevents some code optimizations such as static binding od messages to methods.

Singleton variables

Singleton variables in a clause are often misspell variables and, as such, one of the most common errors when programming in Prolog. when the backend Prolog compiler complies with the Prolog ISO standard or at least supports the ISO predicate `read_term/3` called with the option `singletons(S)`, the Logtalk compiler warns about any singleton variable found while compiling a source file.

Redefinition of Prolog built-in predicates

The Logtalk compiler will warn us of any redefinition of a Prolog built-in predicate inside an object or category. Sometimes the redefinition is intended. In other cases, the user may not be aware that the subjacent Prolog compiler may already provide the predicate as a built-in or we may want to ensure code portability among several Prolog compilers with different sets of built-in predicates.

Redefinition of Logtalk built-in predicates

Similar to the redefinition of Prolog built-in predicates, the Logtalk compiler will warn us if we try to redefine a Logtalk built-in. The redefinition will probably be an error in almost all (if not all) cases.

Redefinition of Logtalk built-in methods

An error will be thrown if we attempt to redefine a Logtalk built-in method inside an entity. The default behavior is to report the error and abort the compilation of the offending entity.

Misspell calls of local predicates

A warning will be reported if Logtalk finds (in the body of a predicate definition) a call to a local predicate that is not defined, built-in (either in Prolog or in Logtalk) or declared dynamic. In most cases these calls are simple misspell errors.

Portability warnings

A warning will be reported if a predicate clause contains a call to a non-ISO specified built-in predicate or arithmetic function, Portability warnings are also reported for non-standard flags or flag values. These warnings often cannot be avoided due to the limited scope of the ISO Prolog standard.

Missing directives

A warning will be reported for any missing dynamic, discontiguous, meta-predicate, and public predicate directive.

Trivial fails

A warning will be reported for any call to a local static predicate with no matching clause.

Redefinition of predicates declared in `uses/2` and `use_module/2` directives

A error will be reported for any attempt to define locally a predicate that is already listed in a `uses/2` or in a `use_module/2` directive.

Other warnings and errors

The Logtalk compiler will throw an error if it finds a predicate clause or a directive that cannot be parsed. The default behavior is to report the error and abort the compilation of the offending entity.

Runtime errors

This section briefly describes runtime errors that result from misuse of Logtalk built-in predicates, built-in methods or from message sending. For a complete and detailed description of runtime errors please consult the Reference Manual.

Logtalk built-in predicates

Most Logtalk built-in predicates checks the type and mode of the calling arguments, throwing an exception in case of misuse.

Logtalk built-in methods

Most Logtalk built-in method checks the type and mode of the calling arguments, throwing an exception in case of misuse.

Message sending

The message sending mechanisms always check if the receiver of a message is a defined object and if the message corresponds to a declared predicate within the scope of the sender. The built-in protocol `forwarding` declares a predicate, `forward/1`, which is automatically called (if defined) by the runtime for any message that the receiving object does not understand. The usual definition for this error handler is to delegate or forward the message to another object that might be able to answer it:

```
forward(Message) :-
    [Object::Message]. % forward the message while preserving the original sender
```

If preserving the original sender is not required, this definition can be simplified to:

```
forward(Message) :-
    Object::Message.
```

More sophisticated definitions are, of course, possible.

Documenting Logtalk programs

By setting the compiler flag `source_data`, Logtalk saves all relevant documenting information collected when compiling a source file. The provided `lgtdoc` tool can access this information by using Logtalk's reflection support and generate a documentation file for each compiled entity (object, protocol, or category) in XML format. Contents of the XML file include the entity name, type, and compilation mode (static or dynamic), the entity relations with other entities, and a description of any declared predicates (name, compilation mode, scope, ...).

The XML documentation files can be enriched with arbitrary user-defined information, either about an entity or about its predicates, by using the two directives described below.

Documenting directives

Logtalk supports two documentation directives for providing arbitrary user-defined information about an entity or a predicate. These two directives complement other Logtalk directives that also provide important documentation information like `mode/2`.

Entity directives

Arbitrary user-defined entity information can be represented using the `info/1` directive:

```
:- info([
    Key1 is Value1,
    Key2 is Value2,
    ...
]).
```

In this pattern, keys should be atoms and values should be ground terms. The following keys are pre-defined and may be processed specially by Logtalk:

`comment`
: Comment describing entity purpose (an atom).

`author`
: Entity author(s) (an atom or a compound term `{entity}` where `entity` is the name of a XML entity defined in the `custom.ent` file).

`version`
: Version number (a number).

`date`
: Date of last modification (formatted as Year/Month/Day).

`parameters`
: Parameter names and descriptions for parametric entities (a list of key-values where both keys and values are atoms).

`parnames`
: Parameter names for parametric entities (a list of atoms; a simpler version of the previous key, used when parameter descriptions are deemed unnecessary).

`copyright`
: Copyright notice for the entity source code (an atom or a compound term `{entity}` where `entity` is the name of a XML entity defined in the `custom.ent` file).

`license`
: License terms for the entity source code; usually, just the license name (an atom or a compound term `{entity}` where `entity` is the name of a XML entity defined in the `custom.ent` file).

`remarks`
: List of general remarks about the entity using the format *Topic* - *Text*. Both the topic and the text must be atoms.

`see_also`
: List of related entities.

For example:

```
:- info([
    version is 2.1,
    author is 'Paulo Moura',
    date is 2000/4/20,
    comment is 'Building representation.',
    diagram is 'UML Class Diagram #312'
]).
```

Use only the keywords that make sense for your application and remember that you are free to invent your own keywords. All key-value pairs can be retrieved programmatically using the reflection API and are visible to `lgtdoc` tool.

Predicate directives

Arbitrary user-defined predicate information can be represented using the `info/2` directive:

```
:- info(Name/Arity, [
    Key1 is Value1,
    Key2 is Value2,
    ...
]).
```

The first argument can also a grammar rule non-terminal indicator, `Name//Arity`. Keys should be atoms and values should be bound terms. The following keys are pre-defined and may be processed specially by Logtalk:

`comment`
: Comment describing predicate purpose (an atom).

`arguments`
: Names and descriptions of predicate arguments for pretty print output (a list of key-values where both keys and values are atoms).

`argnames`
: Names of predicate arguments for pretty print output (a list of atoms; a simpler version of the previous key, used when argument descriptions are deemed unnecessary).

`allocation`
: Objects where we should define the predicate. Some possible values are `container`, `descendants`, `instances`, `classes`, `subclasses`, and `any`.

`redefinition`
: Describes if predicate is expected to be redefined and, if so, in what way. Some possible values are `never`, `free`, `specialize`, `call_super_first`, `call_super_last`.

`exceptions`
: List of possible exceptions throw by the predicate using the format *Description* `-` *Exception term*. The description must be an atom. The exception term must be a non-variable term.

`examples`
: List of typical predicate call examples using the format *Description* `-` *Predicate call* `-` *Variable bindings*. The description must be an atom. The predicate call term must be a non-variable term. The variable bindings term uses the format `{`*Variable*`=`*Term*`, ...}`. When there are no variable bindings, the success or failure of the predicate call should be represented by the terms `{yes}` or `{no}`, respectively.

`remarks`
: List of general remarks about the predicate using the format *Topic* `-` *Text*. Both the topic and the text must be atoms.

For example:

```
:- info(color/1, [
    comment is 'Table of defined colors.',
    argnames is ['Color'],
    constraint is 'Only a maximum of four visible colors allowed.'
]).
```

As with the `info/1` directive, use only the keywords that make sense for your application and remember that you are free to invent your own keywords. All key-value pairs can also be retrieved programmatically using the reflection API and are visible to `lgtdoc` tool.

Processing and viewing documenting files

The `lgtdoc` tool generates a XML documenting file per entity. It can also generate directory, entity, and predicate indexes when documenting libraries and directories. For example, assuming the default filename extensions, a `trace` object and a `sort(_)` parametric object will result in `trace_0.xml` and `sort_1.xml` XML files.

Each entity XML file contains references to two other files, a XML specification file and a XSL style-sheet file. The XML specification file can be either a DTD file (`logtalk_entity.dtd`) or a XML Scheme file (`logtalk_entity.xsd`). The XSL style-sheet file is responsible for converting the XML files to some desired format such as HTML or PDF. The default names for the XML specification file and the XSL style-sheet file are defined by the `lgtdoc` tool but can be overiden by passing a list of options to the tool predicates. The `lgtdoc/xml` sub-directory in the Logtalk installation directory contains the XML specification files described above, along with several sample XSL style-sheet files and sample scripts for converting XML documenting files to several formats (e.g. Markdown, HTML, and PDF). Please read the `NOTES` file included in the directory for details. You may use the supplied sample files as a starting point for generating the documentation of your Logtalk applications.

The Logtalk DTD file, `logtalk_entity.dtd`, contains a reference to a user-customizable file, `custom.ent`, which declares XML entities for source code author names, license terms, and copyright string. After editing the `custom.ent`

file to reflect your personal data, you may use the XML entities on `info/1` documenting directives. For example, assuming that the XML entities are named *author*, *license*, and *copyright* we may write:

```
:- info([
    version is 1.1,
    author is {author},
    license is {license},
    copyright is {copyright}
]).
```

The entity references are replaced by the value of the corresponding XML entity when the XML documenting files are processed (**not** when they are generated; this notation is just a shortcut to take advantage of XML entities).

The `lgtdoc` tool supports a set of options that can be used to control the generation of the XML documentation files. Please see the tool documentation for details.

Inline formatting in comments text

Inline formatting in comments text can be accomplished by using Markdown syntax and converting XML documenting files to Markdown files (and these, if required, to e.g. HTML or PDF).

Installing Logtalk

This page provides an overview of Logtalk installation requirements and instructions and a description of the files contained on the Logtalk distribution. For detailed, up-to-date installation and configuration instructions, please see the `README.md`, `INSTALL.md`, and `CUSTOMIZE.md` files distributed with Logtalk. The broad compatibility of Logtalk, both with Prolog compilers and operating-systems, together with all the possible user scenarios, means that installation can vary from very simple by running an installer or a couple of scripts to the need of patching both Logtalk and Prolog compilers to workaround the lack of strong Prolog standards or to cope with the requirements of less common operating-systems.

The preferred installation scenario is to have Logtalk installed in a system-wide location, thus available for all users, and a local copy of user-modifiable files on each user home directory (even when you are the single user of your computer). This scenario allows each user to independently customize Logtalk and to freely modify the provided libraries and programming examples. Logtalk installers, installation shell scripts, and Prolog integration scripts favor this installation scenario, although alternative installation scenarios are always possible. The installers set two environment variables, `LOGTALKHOME` and `LOGTALKUSER`, pointing, respectively, to the Logtalk installation folder and to the Logtalk user folder.

User applications should preferable be kept outside of the Logtalk user folder created by the installation process, however, as updating Logtalk often results in updating the contents of this folder. If your applications depend on customizations to the distribution files, backup those changes before updating Logtalk.

Hardware and software requirements

Computer and operating system

Logtalk is compatible with almost any computer/operating-system with a modern, standars compliant, Prolog compiler available.

Prolog compiler

Logtalk requires a backend Prolog compiler supporting official and de facto standards. Capabilities needed by Logtalk that are not defined in the official ISO Prolog Core standard include:

- access to predicate properties
- operating-system access predicates
- de facto standard predicates not (yet) specified in the official standard

Logtalk needs access to the predicate property `built_in` to properly compile objects and categories that contain Prolog built-in predicates calls. In addition, some Logtalk built-ins need to know the dynamic/static status of predicates to ensure correct application. The ISO standard for Prolog modules defines a `predicate_property/2` predicate that is already implemented by most Prolog compilers. Note that if these capabilities are not built-in the user cannot easily define them.

For optimal performance, Logtalk requires that the Prolog compiler supports **first-argument indexing** for both static and dynamic code (most modern compilers support this feature).

Since most Prolog compilers are moving closer to the ISO Prolog standard [ISO 95], it is advisable that you try to use the most recent version of your favorite Prolog compiler.

Logtalk installers

Logtalk installers are available for MacOS X, Linux, and Microsoft Windows. Depending on the chosen installer, some tasks (e.g. setting environment variables or integrating Logtalk with some Prolog compilers) may need to be performed manually.

Source distribution

Logtalk sources are available in a `tar` archive compressed with `bzip2`, `lgt3xxx.tar.bz2`. You may expand the archive by using a decompressing utility or by typing the following commands at the command-line:

```
% tar -jxvf lgt3xxx.tar.bz2
```

This will create a sub-directory named `lgt3xxx` in your current directory. Almost all files in the Logtalk distribution are text files. Different operating-systems use different end-of-line codes for text files. Ensure that your decompressing utility converts the end-of-lines of all text files to match your operating system.

Directories and files organization

In the Logtalk installation directory, you will find the following files and directories:

`BIBLIOGRAPHY.bib` – Logtalk bibliography in BibTeX format
`CUSTOMIZE.md` – Logtalk end-user customization instructions
`INSTALL.md` – Logtalk installation instructions
`LICENSE.txt` – Logtalk user license
`NOTICE.txt` – Logtalk copyright notice
`QUICK_START.md` – Quick start instructions for those that do not like to read manuals
`README.md` – several useful information
`RELEASE_NOTES.md` – release notes for this version
`UPGRADING.md` – instructions on how to upgrade your programs to the current Logtalk version
`VERSION.txt` – file containing the current Logtalk version number (used for compatibility checking when upgrading Logtalk)

`loader-sample.lgt` – sample loader file for user applications
`settings-sample.lgt` – sample file for user-defined Logtalk settings
`tester-sample.lgt` – sample file for helping automating running user application unit tests

`adapters`
- `NOTES.md` – notes on the provided adapter files
- `template.pl` – template adapter file
- `...` – specific adapter files

`coding`
- `NOTES.md` – notes on syntax highlighter and text editor support files providing syntax coloring for publishing and editing Logtalk source code

`...` – syntax coloring support files

`contributions`
- `NOTES.md` – notes on the user-contributed code
- `...` – user-contributed code files

`core`
- `NOTES.md` – notes on the current status of the compiler and runtime
- `...` – core source files

`docs`
- `NOTES.md` – notes on the provided documentation for core, library, tools, and contributions entities
- `index.html` – root document for all entities documentation
- `...` – other entity documentation files

`examples`
- `NOTES.md` – short description of the provided examples
- `bricks`
  - `NOTES.md` – example description and other notes
  - `SCRIPT.txt` – step by step example tutorial
  - `loader.lgt` – loader utility file for the example objects
  - `...` – bricks example source files
- `...` – other examples

`integration`
- `NOTES.md` – notes on scripts for Logtalk integration with Prolog compilers
- `...` – Prolog integration scripts

`library`
- `NOTES.md` – short description of the library contents
- `all_loader.lgt` – loader utility file for all library entities
- `...` – library source files

`man`
- `...` – POSIX man pages for the shell scripts

`manuals`
- `NOTES.md` – notes on the provided documentation
- `bibliography.html` – bibliography
- `glossary.html` – glossary
- `index.html` – root document for all documentation
- `...` – other documentation files

`paths`
- `NOTES.md` – description on how to setup library and examples paths
- `paths.pl` – default library and example paths

`scratch`
- `NOTES.md` – notes on the scratch directory

`scripts`
- `NOTES.md` – notes on scripts for Logtalk user setup, packaging, and installation

`...` – packaging, installation, and setup scripts

`tests`
: `NOTES.md` – notes on the current status of the unit tests
`...` – unit tests for built-in features

`tools`
: `NOTES.md` – notes on the provided programming tools
`...` – programming tools

Adapter files

Adapter files provide the glue code between the Logtalk compiler/runtime and a Prolog compiler. Each adapter file contains two sets of predicates: ISO Prolog standard predicates and directives not built-in in the target Prolog compiler and Logtalk-specific predicates.

Logtalk already includes ready to use adapter files for most academic and commercial Prolog compilers. If a adapter file is not available for the compiler that you intend to use, then you need to build a new one, starting from the included `template.pl` file. Start by making a copy of the template file. Carefully check (or complete if needed) each listed definition. If your Prolog compiler conforms to the ISO standard, this task should only take you a few minutes. In most cases, you can borrow code from some of the predefined adapter files. If you are unsure that your Prolog compiler provides all the ISO predicates needed by Logtalk, try to run the system by setting the unknown predicate error handler to report as an error any call to a missing predicate. Better yet, switch to a modern, ISO compliant, Prolog compiler. If you send me your adapter file, with a reference to the target Prolog compiler, maybe I can include it in the next release of Logtalk.

The adapter files specifies *default* values for most of the Logtalk compiler flags. Most of these compiler flags are described in the next section. A few of these flags have read-only values and cannot be changed at runtime. These are:

`settings_file`
: Allows or disables loading of *settings files* at startup. Possible values are `allow`, `restrict`, and `deny`. The usual default value is `allow` but it can be changed by editing the adapter file when e.g. embedding Logtalk in a compiled application. With a value of `allow`, settings files are searched in the startup directory, in the Logtalk

user directory, and in the user home directory. With a value of `restrict`, settings files are only searched in the Logtalk user directory and in the user home directory.

`prolog_dialect`
: Name of the back-end Prolog compiler (an atom). This flag can be used for conditional compilation of Prolog specific code.

`prolog_version`
: Version of the back-end Prolog compiler (a compound term, `(Major, Minor, Patch)`, whose arguments are integers). This flag availability depends on the Prolog compiler. Checking the value of this flag fails for any Prolog compiler that does not provide access to version data.

`prolog_compatible_version`
: Compatible version of the back-end Prolog compiler (a compound term, usually with the format `@>=((Major, Minor, Patch))`, whose arguments are integers). This flag availability depends on the Prolog compiler. Checking the value of this flag fails for any Prolog compiler that does not provide access to version data.

`prolog_conformance`
: Level of conformance of the back-end Prolog compiler with the ISO Prolog Core standard. The possible values are `strict` for compilers claiming strict conformance and `lax` for compilers claiming only broad conformance.

`unicode`
: Informs Logtalk if the back-end Prolog compiler supports the Unicode standard. Possible flag values are `unsupported`, `full` (all Unicode planes supported), and `bmp` (supports only the Basic Multilingual Plane).

`encoding_directive`
: Informs Logtalk if the back-end Prolog compiler supports the `encoding/1` directive. This directive is used for declaring the text encoding of source files. Possible flag values are `unsupported`, `full` (can be used in both Logtalk source files and compiler generated Prolog files), and `source` (can be used only in Logtalk source files).

`tabling`
: Informs Logtalk if the back-end Prolog compiler provides tabling programming support. Possible flag values are `unsupported` and `supported`.

`engines`
: Informs if the back-end Prolog compiler provides the required low level multi-threading programming support for Logtalk threaded engines. Possible flag values are `unsupported` and `supported`.

`threads`
: Informs if the back-end Prolog compiler provides the required low level multi-threading programming support for all high-level Logtalk multi-threading features. Possible flag values are `unsupported` and `supported`.

`modules`
: Informs Logtalk if the back-end Prolog compiler provides suitable module support. Possible flag values are `unsupported` and `supported` (Logtalk provides limited support for compiling Prolog modules as objects).

`coinduction`
: Informs Logtalk if the back-end Prolog compiler provides the minimal support for cyclic terms necessary for working with coinductive predicates. Possible flag values are `unsupported` and `supported`.

Settings files

Although is always possible to edit the back-end Prolog compiler adapter files, the recommended solution to customize compiler flags is to edit the `settings.lgt` file in the Logtalk user folder or in the user home folder. Depending on the back-end Prolog compiler and on the operating-system, is also possible to define per-project settings files by creating a `settings.lgt` file in the project directory and by starting Logtalk from this directory. At startup, Logtalk tries to load a `settings.lgt` file from the startup directory (assuming that the read-only `settings` flag is set to `allow`). If not found, Logtalk tries to load a `settings.lgt` file from the Logtalk user folder. If still not found, Logtalk tries to load a `settings.lgt` file from the user home folder. If no settings files are found, Logtalk will use the default compiler flag values set on the back-end Prolog compiler adapter files. When limitations of the back-end Prolog compiler or on the operating-system prevent Logtalk from finding the settings files, these can always be loaded manually after Logtalk startup.

Settings files are normal Logtalk source files (although when automatically loaded by Logtalk they are compiled silently with any errors being simply ignored). The usual contents is an `initialization/1` Prolog directive containing calls to the `set_logtalk_flag/2` Logtalk built-in predicate and asserting clauses for the `logtalk_library_path/2` multifile dynamic predicate. Note that the `set_logtalk_flag/2` directive cannot be used as its scope is local to the source file being compiled. For example, one of the troubles of writing portable applications is the different feature sets of Prolog compilers. A typical issue is the lack of support for tabling. Using the Logtalk support for conditional compilation you could write:

```
:- if(current_logtalk_flag(tabling, supported)).

    % add tabling directives to the source code
    :- table(foo/1).
    :- table(bar/2).

:- endif.
```

The Logtalk flag `prolog_dialect` may also be used with the conditional compilation directives in order to define a single settings file that can be used with several back-end Prolog compilers. For example:

```
:- if(current_logtalk_flag(prolog_dialect, yap)).

    % YAP specific settings
    ...

:- elif(current_logtalk_flag(prolog_dialect, gnu)).

    % GNU Prolog specific settings
    ...

:- else.

    % generic Prolog settings

:- endif.
```

Logtalk compiler and runtime

The `compiler` sub-directory contains the Prolog source file(s) that implement the Logtalk compiler and the Logtalk runtime. The compiler and the runtime may be split in two (or more) separate files or combined in a single file, depending on the Logtalk release that you are installing.

Library

Starting from version 2.7.0, Logtalk contains a standard library of useful objects, categories, and protocols. Read the corresponding `NOTES.md` file for details about the library contents.

Examples

Logtalk 2.x contains new implementations of some of the examples provided with previous 1.x versions. The sources of each one of these examples can be found included in a subdirectory with the same name, inside the directory examples. The majority of these examples include a file named `SCRIPT.txt` that contains cases of simple utilization. Some examples may depend on other examples and library objects to work properly. Read the corresponding `NOTES.md` file for details before running an example.

Logtalk source files

Logtalk source files are text files containing one or more entity definitions (objects, categories, or protocols). The Logtalk source files may also contain plain Prolog code. The extension `.lgt` is normally used. Logtalk compiles these files to plain Prolog by appending to the file name a suffix derived from the extension and by replacing the `.lgt` extension with `.pl` (`.pl` is the default Prolog extension; if your Prolog compiler expects the Prolog source filenames to end with a specific, different extension, you can set it in the corresponding adapter file).

Writing, running, and debugging applications

Writing applications

For a successful programming in Logtalk, you need a good working knowledge of Prolog and an understanding of the principles of object-oriented programming. Most 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 applications in Logtalk is similar to writing applications 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 define any number of entities (objects, categories, or protocols) and 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 adapter files. Intermediate Prolog source files (generated by the Logtalk compiler) have, by default, a `_lgt` suffix and a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding adapter 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_lgt.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_lgt.pl` Prolog file. This name scheme helps avoid file name conflicts (remember that all Logtalk entities share the same name space). To further prevent file name conflicts, depending on the backend compiler, the names of the intermediate Prolog files may include a directory hash.

Logtalk source files may contain Prolog code interleaved with Logtalk entity definitions. Plain Prolog code is copied as-is to the corresponding Prolog output file (except, of course, if subject to the term-expansion mechanism). Prolog modules are compiled as objects. The following Prolog directives are processed when read (thus affecting the compilation of the source code that follows): `ensure_loaded/1`, `use_module/1-2`, `op/3`, and `set_prolog_flag/2`. The `initialization/1` Prolog directive may be used for defining an initialization goal to be executed when loading a source file.

The text encoding used in a source file may be declared using the `encoding/1` directive when running Logtalk with backend Prolog compilers that support multiple encodings (check the `encoding_directive` flag in the adapter file of your Prolog compiler). The encoding used (and, in the case of a Unicode encoding, any BOM present) in a source file will be used for the intermediate Prolog file generated by the compiler. Logtalk uses the encoding names specified by IANA (in those cases where a preferred MIME name alias is specified, the alias is used instead).

Logtalk source files can include the text of other files by using the `include/1` directive. Although there's also a standard Prolog `include/1` directive, any occurrences of this directive in a Logtalk source file is handled by the Logtalk compiler, not by the backend Prolog compiler.

Portable applications

Logtalk is compatible with almost all modern Prolog compilers. However, this does not necessarily imply that your Logtalk applications will have the same level of portability. If possible, you should only use in your applications Logtalk built-in predicates and ISO Prolog specified built-in predicates and arithmetic functions. If you need to use built-in predicates (or built-in arithmetic functions) 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 specified built-in predicates and arithmetic functions by using the `portability/1` compiler flag.

Conditional compilation

Logtalk supports conditional compilation within source files using the `if/1`, `elif/1`, `else/0`, and `endif/0` directives. This support is similar to the support found in some Prolog compilers such as ECLiPSe, SWI-Prolog, or YAP.

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-in predicates 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 compiler may not be able to detect misspell variables in your source code.

Coding style guidelines

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 tabs (the recommended setting is a tab width equivalent to 4 spaces) for laying out code. Closed related entities can be defined in the same source file. However, for best performance, is often necessary to have an entity per source file. Entities that might be useful in different contexts (such as library entities) are best defined in their own source files.

Running a Logtalk session

We run Logtalk inside a normal Prolog session, after loading the necessary files. Logtalk extends but does not modify your Prolog compiler. We can freely mix Prolog queries with the sending of messages and our applications can be made of both normal Prolog clauses and object definitions.

Starting Logtalk

Depending on your Logtalk installation, you may use a script or a shortcut to start Logtalk with your chosen Prolog compiler. On POSIX operating systems, the scripts should be available from the command-line; scripts are named upon the used Prolog compilers. On Windows, the shortcuts should be available from the Start Menu. If no scripts or shortcuts are available for your installation, operating-system, or Prolog compiler, you can always start a Logtalk session by performing the following steps:

1. Start your Prolog compiler.
2. Load the appropriate adapter file for your compiler. Adapter files for most common Prolog compilers can be found in the `adapters` subdirectory.
3. Load the library paths file corresponding to your Logtalk installation contained in the `paths` subdirectory.
4. Load the Logtalk compiler/runtime files contained in the `compiler` subdirectory.

Note that the adapter files, compiler/runtime files, and library paths file are Prolog source files. The predicate called to load (and compile) them depends on your Prolog compiler. In case of doubt, consult your Prolog compiler reference manual or take a look at the definition of the predicate `'$lgt_load_prolog_code'/3` in the corresponding adapter file.

Most Prolog compilers support automatic loading of an initialization file, which can include the necessary directives to load both the Prolog adapter file and the Logtalk compiler. This feature, when available, allows automatic loading of Logtalk when you start your Prolog compiler.

Compiling and loading your applications

Your applications will be made of source files containing your objects, protocols, and categories. The source files can be compiled to disk by calling the Logtalk built-in predicate `logtalk_compile/1`:

```
| ?- logtalk_compile([source_file1, source_file2, ...]).
```

This predicate runs the compiler on each file and, if no fatal errors are found, outputs Prolog source files that can then be consulted or compiled in the usual way by your Prolog compiler.

To compile to disk and also load into memory the source files we can use the Logtalk built-in predicate `logtalk_load/1`:

```
| ?- logtalk_load([source_file1, source_file2, ...]).
```

This predicate works in the same way of the predicate `logtalk_compile/1` but also loads the compiled files into memory.

Both predicates expect a source file name or a list of source file names as an argument. The Logtalk source file name extension, as defined in the adapter file (by default, `.lgt`), can be omitted.

If you have more than a few source files then you may want to use a loader helper file containing the calls to the `logtalk_load/1-2` predicates. Consulting or compiling the loader file will then compile and load all your Logtalk entities into memory (see below for details).

With most Prolog back-end compilers, you can use the shorthands `{File}` for `logtalk_load(File)` and `{File1, File2, ...}` for `logtalk_load([File1, File2, ...])`. The use these shorthands should be restricted to the Logtalk/Prolog top-level interpreter as they are not part of the language specification and may be commented out in case of conflicts with backend Prolog compiler features.

The built-in predicate `logtalk_make/0` can be used to reload all modified source files. Files are also reloaded when the compilation mode changes. For example, assume that you have loaded your application files and found a bug. You can easily recompile the files in debug mode by using the queries:

```
| ?- set_logtalk_flag(debug, on).
...

| ?- logtalk_make.
...
```

After debugging and fixing the bugs, you can reload the files in normal (or optimized) mode by turning the debug flag off and calling the `logtalk_make/0` predicate again.

An extended version of this predicate, `logtalk_make/1`, accepts `all`, `clean`, `check`, `circular`, and `documentation` arguments for, respectively, reloading modified Logtalk source files, deleting any intermediate files generated by the compilation of Logtalk source files, checking for code issues, listing of circular dependencies, and generating documentation. With most Prolog backend compilers, you can use the shorthands `{*}` for `logtalk_make(all)`, `{!}` for `logtalk_make(clean)`, `{?}` for `logtalk_make(missing)`, `{@}` for `logtalk_make(circular)`, and `{#}` for `logtalk_make(documentation)`. The `logtalk_make(clean)` goal can be specially useful before switching backend Prolog compilers as the generated intermediate files may not be compatible.

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 ordinary source file and thus compiled and loaded like any 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 `set_logtalk_flag/2` (e.g. for setting global, *project-specific*, flag values) and `logtalk_load/1` or `logtalk_load/2` (for loading project files), wrapped inside a Prolog `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((
    % set project-specific global flags
    set_logtalk_flag(events, allow),
    % load the project source files
    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 source file-specific compiler flags (this is useful even when you only have a single source file if you always load it with using the same set of compiler flags). For example:

```
:- initialization((
    % set project-specific global flags
    set_logtalk_flag(underscore_variables, dont_care),
    set_logtalk_flag(source_data, off),
    % load the project source files
    logtalk_load(
        [source1, source2, source3],
        [portability(warning)]),              % source file-specific flags
    logtalk_load(
        [source4, source5],
        [portability(silent)])                % source file-specific flags
)).
```

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

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

```
| ?- logtalk_load(loader, [underscore_variables(dont_care), source_data(off)]).
```

This will not work as you might expect as the compiler flags 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 defining or asserting clauses for the dynamic and multifile predicate `logtalk_library_path/2`. For example:

```
:- multifile(logtalk_library_path/2).
:- dynamic(logtalk_library_path/2).

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:

```
:- multifile(logtalk_library_path/2).
:- dynamic(logtalk_library_path/2).

logtalk_library_path(lgtuser, '$LOGTALKUSER/').
logtalk_library_path(examples, lgtuser('examples/')).
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 `viewpoints`, 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 clauses for the `logtalk_library_path/2` predicate 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 should always end with the path directory separator character. Most back-end Prolog compilers allows the use of environment variables in the second argument of the `logtalk_library_path/2` predicate. Use of POSIX relative paths (e.g. `'../'` or `'./'`) for top-level library directories (e.g. `lgtuser` in the example above) is not advised as different back-end Prolog compilers may start with different initial working directories, which may result in portability problems of your loader files.

The library notation provides functionality inspired by the `file_search_path/2` mechanism introduced by Quintus Prolog and later adopted by some other Prolog compilers.

Compiler linter

The compiler includes a linter that checks for a wide range of possible problems in source files. Notably, the compiler checks for unknown entities, unknown predicates, undefined predicates (i.e. predicates that are declared but not defined), missing directives (including missing `dynamic/1` and `meta_predicate/1` directives), redefined built-in predicates, calls to non-portable predicates, singleton variables, tautology and falsehood goals (i.e. goals that are can be replaced by `true` or `fail`), and trivial fails (i.e. calls to predicates with no match clauses). Some of the linter warnings are controlled by compiler flags. See the next section for details.

Compiler flags

The `logtalk_load/1` and `logtalk_compile/1` always use the current set of default compiler flags as specified in your settings file and the Logtalk adapter files or changed for the current session using the built-in predicate `set_logtalk_flag/2`. Although the default flag values cover the usual cases, you may want to use a different set of flag values while compiling or loading some of your Logtalk source files. This can be accomplished by using the `logtalk_load/2` or the `logtalk_compile/2` built-in predicates. These two predicates accept a list of options affecting how a Logtalk source file is compiled and loaded:

```
| ?- logtalk_compile(Files, Options).
```

or:

```
| ?- logtalk_load(Files, Options).
```

In fact, the `logtalk_load/1` and `logtalk_compile/1` predicates are just shortcuts to the extended versions called with the default compiler flag values. The options are represented by a compound term where the functor is the flag name and the sole argument is the flag value.

We may also change the default flag values from the ones loaded from the adapter file by using the `set_logtalk_flag/2` built-in predicate. For example:

```
| ?- set_logtalk_flag(unknown_entities, silent).
```

The current default flags values can be enumerated using the `current_logtalk_flag/2` built-in predicate:

```
| ?- current_logtalk_flag(unknown_entities, Value).

Value = silent
yes
```

Logtalk also implements a `set_logtalk_flag/2` directive, which can be used to set flags within a source file or within an entity. For example:

```
% compile all objects in the source file with event support
:- set_logtalk_flag(events, allow).

:- object(foo).

    % compile this object with support for dynamic predicate declarations
    :- set_logtalk_flag(dynamic_declarations, allow).
    ...

:- end_object.

...
```

Note that the scope of the `set_logtalk_flag/2` directive is local to the entity or to the source file containing it.

Version flags

`version_data(Value)`
: Read-only flag whose value is the compound term `logtalk(Major,Minor,Patch,Status)`. The first three arguments are integers and the last argument is an atom, possibly empty, representing version status: `aN` for alpha versions, `bN` for beta versions, `rcN` for release candidates (with `N` being a natural number), and `stable` for stable versions. The `version_data` flag is also a de facto standard for Prolog compilers.

`version(Value)`
: Deprecated read-only flag, inherited from Logtalk 2.x, whose value is the compound term `version(Major,Minor,Patch)`. The arguments are integers. New applications that are not required to support Logtalk 2.x should use the `version_data` flag instead.

Lint flags

`unknown_entities(Option)`
: Controls the unknown entity warnings, resulting from loading an entity that references some other entity that is not currently loaded. Possible option values are `warning` (the usual default) and `silent`. Note that these warnings are not always avoidable, specially when using reflective designs of class-based hierarchies.

`unknown_predicates(Option)`
: Defines the compiler behavior when calls to unknown predicates (or non-terminals) are found. An unknown predicate is a called predicate that is neither locally declared or defined. Possible option values are `error`, `warning` (the usual default), and `silent` (not recommended).

`undefined_predicates(Option)`
: Defines the compiler behavior when calls to declared but undefined predicates (or non-terminals) are found. Note that calls to declared but undefined predicates (or non-terminals) fail as per closed-world assumption. Possible option values are `error`, `warning` (the usual default), and `silent` (not recommended).

`portability(Option)`
: Controls the non-ISO specified Prolog built-in predicate and non-ISO specified Prolog built-in arithmetic function calls warnings plus use of non-standard Prolog flags and/or flag values. Possible option values are `warning` and `silent` (the usual default).

`missing_directives(Option)`
: Controls the missing predicate directive warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`redefined_built_ins(Option)`
: Controls the Logtalk and Prolog built-in predicate redefinition warnings. Possible option values are `warning` (the usual default) and `silent`. Warnings about redefined Prolog built-in predicates are often the result of running a Logtalk application on several Prolog compilers as each Prolog compiler defines its set of built-in predicates.

`singleton_variables(Option)`
: Controls the singleton variable warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`underscore_variables(Option)`
: Controls the interpretation of variables that start with an underscore (excluding the anonymous variable) that occur once in a term as either don't care variables or singleton variables. Possible option values are `dont_care` and `singletons` (the usual default). Note that, depending on your Prolog compiler, the `read_term/3` built-in predicate may report variables that start with an underscore as singleton variables. There is no standard behavior, hence this option.

Optional features compilation flags

`complements(Option)`
: Allows objects to be compiled with support for complementing categories turned off in order to improve performance and security. Possible option values are `allow` (allow complementing categories to override local object predicate declarations and definitions), `restrict` (allow complementing categories to add predicate declarations and definitions to an object but not to override them), and `deny` (ignore complementing categories;

the usual default). This option can be used on a per-object basis. Note that changing this option is of no consequence for objects already compiled and loaded.

`dynamic_declarations(Option)`
: Allows objects to be compiled with support for dynamic declaration of new predicates turned off in order to improve performance and security. Possible option values are `allow` and `deny` (the usual default). This option can be used on a per-object basis. Note that changing this option is of no consequence for objects already compiled and loaded. This option is only checked when sending an `asserta/1` or `assertz/1` message to an object. Local asserting of new predicates is always allowed.

`events(Option)`
: Allows message sending calls to be compiled with event-driven programming support disable in order to improve performance. Possible option values are `allow` and `deny` (the usual default). Objects (and categories) compiled with this option set to `deny` use optimized code for message-sending calls that does not trigger events. As such, this option can be used on a per-object (or per-category) basis. Note that changing this option is of no consequence for objects already compiled and loaded.

`context_switching_calls(Option)`
: Allows context switching calls (`<</2`) to be either allowed or denied. Possible option values are `allow` and `deny`. The default flag vale is `allow`. Note that changing this option is of no consequence for objects already compiled and loaded.

Back-end Prolog compiler and loader flags

`prolog_compiler(Flags)`
: List of compiler flags for the generated Prolog files. The valid flags are specific to the used Prolog backend compiler. The usual default is the empty list. These flags are passed to the backend Prolog compiler built-in predicate that is responsible for compiling to disk a Prolog file. For Prolog compilers that don't provide separate predicates for compiling and loading a file, use instead the `prolog_loader/1` flag.

`prolog_loader(Flags)`
: List of loader flags for the generated Prolog files. The valid flags are specific to the used Prolog backend compiler. The usual default is the empty list. These flags are passed to the backend Prolog compiler built-in predicate that is responsible for loading a (compiled) Prolog file.

Other flags

`scratch_directory(Directory)`
: Sets the directory to be used to store the temporary files generated when compiling Logtalk source files. This directory can be specified using an atom or using library notation. The directory must always end with a slash. The default value is a sub-directory of the source files directory, either `'./lgt_tmp/'` or `'./.lgt_tmp/'` (depending on the back-end Prolog compiler and operating-system). Relative directories must always start with `'./'` due to the lack of a portable solution to check if a path is relative or absolute.

`report(Option)`
: Controls the default printing of messages. Possible option values are `on` (by usual default, print all messages that are not intercepted by the user), `warnings` (only print warning and error messages that are not intercepted by the user), and `off` (do not print any messages that are not intercepted by the user).

`code_prefix(Character)`
: Enables the definition of prefix for all functors of Prolog code generated by the Logtalk compiler. The option value must be a single character atom. Its default value is `'$'`. Specifying a code prefix provides a way to solve possible conflicts between Logtalk compiled code and other Prolog code. In addition, some Prolog compilers

automatically hide predicates whose functor start with a specific prefix such as the character `$`. Although this is not a read-only flag, it should only be changed at startup time and before loading any source files.

`optimize(Option)`
Controls the compiler optimizations. Possible option values are `on` (used by default for deployment) and `off` (used by default for development). Compiler optimizations include the use of static binding whenever possible, the removal of redundant calls to `true/0` from predicate clauses, the removal of redundant unifications when compiling grammar rules, and inlining of predicate definitions with a single clause that links to a local predicate, to a plain Prolog built-in (or foreign) predicate, or to a Prolog module predicate with the same arguments. Care should be taken when developing applications with this flag turned on as changing and reloading a file may render static binding optimizations invalid for code defining in other loaded files. Turning on this flag automatically turns off the `debug` flag.

`source_data(Option)`
Defines how much information is retained when compiling a source file. Possible option values are `on` (the usual default for development) and `off`. With this flag set to `on`, Logtalk will keep the information represented using documenting directives plus source location data (including source file names and line numbers). This information can be retrieved using reflection and is useful for documenting, debugging, and integration with third-party development tools. This flag can be turned off in order to generate more compact code.

`debug(Option)`
Controls the compilation of source files in debug mode (the Logtalk default debugger can only be used with files compiled in this mode). Also controls, by default, printing of `debug>` and `debug(Topic)` messages. Possible option values are `on` and `off` (the usual default). Turning on this flag automatically turns off the `optimize` flag.

`reload(Option)`
Defines the reloading behavior for source files. Possible option values are `skip` (skip loading of already loaded files; this value can be used to get similar functionality to the Prolog directive `ensure_loaded/1` but should be used only with fully debugged code), `changed` (the usual default; reload files only when they are changed since last loaded provided that the any explicit flags and the compilation mode are the same as before), and `always` (always reload files).

`relative_to(Directory)`
Defines a base directory for resolving relative source file paths. The default value is the directory of the source file being compiled.

`hook(Object)`
Allows the definition of compiler hooks that are called for each term read form a source file and for each compiled goal. This option specifies an object (which can be the pseudo-object `user`) implementing the `expanding` built-in protocol. The hook object must be compiled and loaded when this option is used. It's also possible to specify a Prolog module instead of a Logtalk object but the module must be pre-loaded and its identifier must be different from any object identifier. The object is expected to define clauses for the `term_expansion/2` and `goal_expansion/2` predicates. In the case of the `term_expansion/2` predicate, the first argument is the term read form the source file while the second argument returns a list of terms corresponding to the expansion of the first argument. In the case of the `goal_expansion/2` predicate, the second argument should be a goal resulting from the expansion of the goal in the first argument. The predicate

`goal_expansion/2` is recursively called on the expanded goal until a fixed point is reached. Care must be taken to avoid compilation loops.

`clean(Option)`
: Controls cleaning of the intermediate Prolog files generated when compiling Logtalk source files. Possible option values are `off` and `on` (the usual default). When turned on, this flag also forces recompilation of all source files, disregarding any existing intermediate files. Thus, it is strong advisable to turn on this flag when switching backend Prolog compilers as the intermediate files generated by the compilation of source files may not be portable (due to differences in the implementation of the standard `write_canonical/2` predicate).

User-defined flags

Logtalk provides a `create_logtalk_flag/3` predicate that can be used for defining new flags.

Reloading and smart compilation of source files

As a general rule, reloading source files should never occur in production code and should be handled with care in development code. Reloading a Logtalk source file usually requires reloading the intermediate Prolog file that is generated by the Logtalk compiler. The problem is that there is no standard behavior for reloading Prolog files. For static predicates, almost all Prolog compilers replace the old definitions with the new ones. However, for dynamic predicates, the behavior depends on the Prolog compiler. Most compilers replace the old definitions but some of them simply append the new ones, which usually leads to trouble. See the compatibility notes for the back-end Prolog compiler you intend to use for more information. There is an additional potential problem when using multi-threading programming. Reloading a threaded object does not recreate from scratch its old message queue, which may still be in use (e.g. threads may be waiting on it).

When using library entities and stable code, you can avoid reloading the corresponding source files (and, therefore, recompiling them) by setting the compiler flag `reload` to `skip`. For code under development, you can turn off the `clean` flag to avoid recompiling files that have not been modified since last compilation (assuming that back-end Prolog compiler that you are using supports retrieving of file modification dates). You can disable deleting the intermediate files generated when compiling source files by changing the default flag value in your settings file, by using the corresponding compiler flag with the compiling and loading built-in predicates, or, for the remaining of a working session, by using the call:

```
| ?- set_logtalk_flag(clean, off).
```

Some caveats that you should be aware. First, some warnings that might be produced when compiling a source file will not show up if the corresponding object file is up-to-date because the source file is not being (re)compiled. Second, if you are using several Prolog compilers with Logtalk, be sure to perform the first compilation of your source files with smart compilation turned off: the intermediate Prolog files generated by the Logtalk compiler may be not compatible across Prolog compilers or even for the same Prolog compiler across operating systems (e.g. due to the use of different character encodings or end-of-line characters).

Using Logtalk for batch processing

If you use Logtalk for batch processing, you probably want to turn off the option `report` to suppress all messages of type `banner`, `comment`, `comment(_)`, `warning`, and `warning(_)` that are normally printed. Note that error messages and messages providing information requested by the user will still be printed.

Optimizing performance

The default compiler flag settings are appropriated for the **development** but not necessarily for the **deployment** of applications. To minimize the generated code size, turn the `source_data` flag off. To optimize runtime performance, turn on the `optimize` flag.

Pay special attention to file compilation/loading order. Whenever possible, compile/load your files taking into account file dependencies to enable static binding optimizations. The easiest way to find the dependencies and thus the best compilation/loading order is to use the `diagrams` tool to generate a file dependency diagram for your application.

Minimize the use of dynamic predicates. Parametric objects can often be used in alternative. When dynamic predicates cannot be avoided, try to make them private. Declaring a dynamic predicate also as a private predicate allows the compiler to optimize local calls to the database methods (e.g. `assertz/1` and `retract/1`) that handle the predicate.

Sending a message to *self* implies dynamic binding but there are often cases where `::/1` is misused to call an imported or inherited predicate that is never going to be redefined in a descendant. In these cases, a *super* call, `^^/1`, can be used instead with the benefit of enabling static binding. Most of the guidelines for writing efficient Prolog code also apply to Logtalk code. In particular, define your predicates to take advantage of first-argument indexing. In the case of recursive predicates, define them as tail-recursive predicates whenever possible.

Debugging Logtalk applications

The Logtalk distribution includes in its `tools` directory a command-line debugger, implemented as a Logtalk application. It can be loaded by typing:

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

This tool implements debugging features similar to those found on most Prolog systems. There are some differences, however, between the usual implementation of Prolog debuggers and the current implementation of the Logtalk debugger that you should be aware. First, unlike some Prolog debuggers, the Logtalk debugger is not implemented as a meta-interpreter. This translates to a different, although similar, set of debugging features when compared with some of the more sophisticated Prolog debuggers. Second, debugging is only possible for entities compiled in debug mode. When compiling an entity in debug mode, Logtalk decorates clauses with source information to allow tracing of the goal execution. Third, implementation of spy points allows the user to specify the execution context for entering the debugger. This feature is a consequence of the encapsulation of predicates inside objects.

Compiling source files and entities in debug mode

Compilation of source files in debug mode is controlled by the compiler flag `debug`. The default value for this flag, usually `off`, is defined in the adapter files. Its value may be changed at runtime by writing:

```
| ?- set_logtalk_flag(debug, on).
```

In alternative, if we want to compile only some source files in debug mode, we may instead write:

```
| ?- logtalk_load([file1, file2, ...], [debug(on)]).
```

The compiler flag `clean` should be turned on whenever the debug flag is turned on at runtime. This is necessary because debug code would not be generated for files previously compiled in normal mode if there are no changes to the source files.

After loading the debugger, we may check or enumerate, by backtracking, all loaded entities compiled in debug mode as follows:

```
| ?- debugger::debugging(Entity).
```

To compile only a specific entity in debug mode, use the `set_logtalk_flag/2` directive inside the entity.

Logtalk Procedure Box model

Logtalk uses a *Procedure Box model* similar to those found on most Prolog compilers. The traditional Prolog procedure box model uses four ports (*call*, *exit*, *redo*, and *fail*) for describing control flow when a predicate clause is used during program execution:

`call`
: predicate call

`exit`
: success of a predicate call

`redo`
: backtracking into a predicate

`fail`
: failure of a predicate call

Logtalk, as found on some recent Prolog compilers, adds a port for dealing with exceptions thrown when calling a predicate:

`exception`
: predicate call throws an exception

In addition to the ports described above, Logtalk adds two more ports, *fact* and *rule*, which show the result of the unification of a goal with, respectively, a fact and a rule head:

`fact`
: unification success between a goal and a fact

`rule`
: unification success between a goal and a rule head

The user may define for which ports the debugger should pause for user interaction by specifying a list of leashed ports. For example:

```
| ?- debugger::leash([call, exit, fail]).
```

Alternatively, the user may use an atom abbreviation for a pre-defined set of ports. For example:

```
| ?- debugger::leash(loose).
```

The abbreviations defined in Logtalk are similar to those defined on some Prolog compilers:

```
none
     []
loose
     [fact, rule, call]
half
     [fact, rule, call, redo]
tight
     [fact, rule, call, redo, fail, exception]
full
     [fact, rule, call, exit, redo, fail, exception]
```

By default, the debugger pauses at every port for user interaction.

Defining spy points

Logtalk spy points can be defined by simply stating which file line numbers or predicates should be spied, as in most Prolog debuggers, or by fully specifying the context for activating a spy point. In the case of line number spy points, the line number must correspond to the first line of an entity clause. To simplify the definition of line number spy points, these are specified using the entity identifier instead of the file name (as all entities share a single namespace, an entity can only be defined in a single file).

Defining line number and predicate spy points

Line number and predicate spy points are specified using the method `spy/1`. The argument can be either a pair entity identifier - line number (`Entity-Line`) or a predicate indicator (`Name/Arity`) or a list of spy points. For example:

```
| ?- debugger::spy(person-42).

Spy points set.
yes

| ?- debugger::spy(foo/2).

Spy points set.
yes

| ?- debugger::spy([foo/4, bar/1]).

Spy points set.
yes
```

Line numbers and predicate spy points can be removed by using the method `nospy/1`. The argument can be a spy point, a list of spy points, or a non-instantiated variable in which case all spy points will be removed. For example:

```
| ?- debugger::nospy(_).

All matching predicate spy points removed.
yes
```

Defining context spy points

A context spy point is a term describing a message execution context and a goal:

```
(Sender, This, Self, Goal)
```

The debugger is evoked whenever the execution context is true and when the spy point goal unifies with the goal currently being executed. Variable bindings resulting from the unification between the current goal and the goal argument are discarded. The user may establish any number of context spy points as necessary. For example, in order to call the debugger whenever a predicate defined on an object named `foo` is called we may define the following spy point:

```
| ?- debugger::spy(_, foo, _, _).

Spy point set.
yes
```

For example, we can spy all calls to a `foo/2` predicate by setting the condition:

```
| ?- debugger::spy(_, _, _, foo(_, _)).

Spy point set.
yes
```

The method `nospy/4` may be used to remove all matching spy points. For example, the call:

```
| ?- debugger::nospy(_, _, foo, _).

All matching context spy points removed.
yes
```

will remove all context spy points where the value of *Self* matches the name `foo`.

Removing all spy points

We may remove all line number, predicate, and context spy points by using the method `nospyall/0`:

```
| ?- debugger::nospyall.

All line number spy points removed.
All predicate spy points removed.
All context spy points removed.
yes
```

Tracing program execution

Logtalk allows tracing of execution for all objects compiled in debug mode. To start the debugger in trace mode, write:

```
| ?- debugger::trace.

yes
```

Note that, when tracing, spy points will be ignored. While tracing, the debugger will pause for user input at each leashed port, printing an informative message with the port name and the current goal. Before the port number, when a spy point is set for the current clause or goal, the debugger will print a `#` character for line number spy points, a `+` character for predicate spy points, and a `*` character for context spy points. The debugger also provides determinism information by prefixing the `exit` port with a `*` character when a call succeeds with choice-points pending. After the port name, the debugger prints the goal invocation number. This invocation number is unique and can be used to correlate the port trace messages.

To stop tracing and turning off the debugger, write:

```
| ?- debugger::notrace.

yes
```

Debugging using spy points

Tracing a program execution may generate large amounts of debugging data. Debugging using spy points allows the user to concentrate its attention in specific points of its code. To start a debugging session using spy points, write:

```
| ?- debugger::debug.

yes
```

At the beginning of a port description, the debugger will print a `#`, `+`, or `*` character before the current goal if there is, respectively, a line number, a predicate, or a context spy point defined.

To stop the debugger, write:

```
| ?- debugger::nodebug.

yes
```

Note that stopping the debugger does not remove any defined spy points.

Debugging commands

The debugger pauses at leashed ports when tracing or when finding a spy point for user interaction. The commands available are as follows:

`c` — creep
: go on; you may use the spacebar, return, or enter keys in alternative

`l` — leap
: continues execution until the next spy point is found

`s` — skip
: skips debugging for the current goal; valid at call, redo, and unification ports

`q` — quasi-skip
: skips debugging until returning to the current goal or reaching a spy point; valid at call and redo ports

`r` — retry
: retries the current goal but side-effects are not undone; valid at the fail port

`j` — jump
: reads invocation number and continues execution until a port is reached for that number

`z` — zap
: reads port name and continues execution until that port is reached
: reads negated port name and continues execution until a port other than the negated port is reached

`i` — ignore
: ignores goal, assumes that it succeeded; valid at call and redo ports

`f` — fail
: forces backtracking; may also be used to convert an exception into a failure

`n` — nodebug
: turns off debugging

`@` — command; `!` can be used in alternative
: reads and executes a query

`b` — break
: suspends execution and starts new interpreter; type `end_of_file` to terminate

`a` — abort
: returns to top level interpreter

`Q` — quit
: quits Logtalk

`p` — print
: writes current goal using the print/1 predicate if available

`d` — display
: writes current goal without using operator notation

`w` — write
: writes current goal quoting atoms if necessary

`$` — dollar
: outputs the compiled form of the current goal (for low-level debugging)

`x` — context
: prints execution context

`.` — file
: prints file, entity, predicate, and line number information at an unification port

`e` — exception
: prints exception term thrown by the current goal

`=` — debugging
: prints debugging information

`<` — write depth
: sets the write term depth (set to 0 to reset)

`*` — add
: adds a context spy point for the current goal

`/` — remove
: removes a context spy point for the current goal

`+` — add
: adds a predicate spy point for the current goal

`-` — remove
: removes a predicate spy point for the current goal

`#` — add
: adds a line number spy point for the current clause

`|` — remove
: removes a line number spy point for the current clause

`h` — condensed help
: prints list of command options

`?` — extended help
: prints list of command options

Context-switching calls

Logtalk provides a control construct, `<</2`, which allows the execution of a query within the context of an object. Common debugging uses include checking an object local predicates (e.g. predicates representing internal dynamic state) and sending a message from within an object. This control construct may also be used to write unit tests.

Consider the following toy example:

```
:- object(broken).

    :- public(a/1).
    :- private([b/2, c/1]).
    :- dynamic(c/1).

    a(A) :- b(A, B), c(B).
    b(1, 2). b(2, 4). b(3, 6).
    c(3).    % in a real-life example, this would be a clause asserted at runtime

:- end_object.
```

Something is wrong when we try the object public predicate, `a/1`:

```
| ?- broken::a(A).

no
```

For helping diagnosing the problem, instead of compiling the object in debug mode and doing a *trace* of the query to check the clauses for the non-public predicates, we can instead simply type:

```
| ?- broken << c(C).

C = 3
yes
```

The `<</2` control construct works by switching the execution context to the object in the first argument and then compiling and executing the second argument within that context:

```
| ?- broken << (self(Self), sender(Sender), this(This)).

Self = broken
Sender = broken
This = broken

yes
```

As exemplified above, the `<</2` control construct allows you to call an object local and private predicates. However, it is important to stress that we are not bypassing or defeating an object predicate scope directives. The calls take place within the context of the specified object, not within the context of the object making the `<</2` call. Thus, the `<</2` control construct implements a form of *execution-context switching*.

The availability of the `<</2` control construct is controlled by the compiler flag `context_switching_calls` (its default value is defined in the adapter files of the back-end Prolog compilers).

Using compilation hooks and term expansion for debugging

It is possible to use compilation hooks and the term expansion mechanism for conditional compilation of debugging goals. Assume that we chose the predicate `debug/1` to represent debug goals. For example:

```
append([], List, List) :-
    debug((write('Base case: '), writeq(append([], List, List)), nl)).
append([Head| Tail], List, [Head| Tail2]) :-
    debug((write('Recursive case: '), writeq(append(Tail, List, Tail2)), nl)),
    append(Tail, List, Tail2).
```

When debugging, we want to call the argument of the predicate `debug/1`. This can be easily accomplished by defining a hook object containing the following definition for `goal_expansion/2`:

```
goal_expansion(debug(Goal), Goal).
```

When not debugging, we can use a second hook object to discard the `debug/1` calls by defining the predicate `goal_expansion/2` as follows:

```
goal_expansion(debug(_), true).
```

The Logtalk compiler automatically removes any redundant calls to the built-in predicate `true/0` when compiling object predicates.

Debugging messages

Calls to the `logtalk::print_message/3` predicate where the message kind is either `debug` or `debug(_)` are only printed, by default, when the `debug` flag is turned on. Note that using these messages does not require compiling the code

in debug mode, only turning on the flag. To avoid having to define `message_tokens//2` grammar rules for translating each debug message, Logtalk provides default tokenization for four *meta-messages* that cover the most common cases:

`@Message`
: By default, the message is printed as passed to the `write/1` predicate followed by a newline.

`Key-Value`
: By default, the message is printed as `Key: Value` followed by a newline. The value is printed as passed to the `writeq/1` predicate.

`List`
: By default, the list items are printed indented one per line. The items are preceded by a dash and printed as passed to the `writeq/1` predicate.

`Title::List`
: By default, the title is printed followed by a newline and the indented list items, one per line. The items are preceded by a dash and printed as passed to the `writeq/1` predicate.

These print messages goals can always be combined with hooks as described in the previous section to remove them in production ready code.

Some simple examples of using these meta-messages:

```
| ?- logtalk::print_message(debug, core, @'Phase 1 completed').
yes

| ?- set_logtalk_flag(debug, on).
yes

| ?- logtalk::print_message(debug, core, @'Phase 1 completed').
>>> Phase 1 completed
yes

| ?- logtalk::print_message(debug, core, answer-42).
>>> answer: 42
yes

| ?- logtalk::print_message(debug, core, ['Arthur Dent','Ford Prefect','Marvin']).
>>> - 'Arthur Dent'
>>> - 'Ford Prefect'
>>> - 'Marvin'
yes

| ?- logtalk::print_message(debug, core, 'Names'::['Arthur Dent','Ford Prefect','Marvin']).
>>> Names:
>>> - 'Arthur Dent'
>>> - 'Ford Prefect'
>>> - 'Marvin'
yes
```

Prolog Integration and Migration Guide

An application may include plain Prolog files, Prolog modules, and Logtalk objects. This is a perfectly valid way of developing a complex application and, in some cases, it might be the most appropriated solution. Modules may be used for legacy code or when a simple encapsulation mechanism is adequate. Logtalk objects may be used when more powerful encapsulation, abstraction, and reuse features are necessary. Logtalk supports the compilation of source files containing both plain Prolog and Prolog modules. This guide provides tips for helping integrating and migrating plain Prolog code and Prolog module code to Logtalk. Step-by-step instructions are provided for encapsulating plain Prolog code in objects, converting Prolog modules into objects, and compiling and reusing Prolog modules as objects from inside Logtalk. An interesting application of the techniques described in this guide is a solution for running a Prolog application which uses modules on a Prolog compiler with no module system.

Source files with both Prolog code and Logtalk code

Logtalk source files may contain plain Prolog code intermixed with Logtalk code. The Logtalk compiler simply copies the plain Prolog code as-is to the generated Prolog file. With Prolog modules, it is assumed that the module code starts with a `module/1-2` directive and ends at the end of the file. There is no module ending directive which would allowed us to define more than one module per file. In fact, most if not all Prolog module systems always define a single module per file. Some of them mandate that the `module/1-2` directive be the first term on a source file. As such, when the Logtalk compiler finds a `module/1-2` directive, it assumes that all code that follows until the end of the file belongs to the module.

Encapsulating plain Prolog code in objects

Most applications consist of several plain Prolog source files, each one defining a few top-level predicates and auxiliary predicates that are not meant to be directly called by the user. Encapsulating plain Prolog code in objects allows us to make clear the different roles of each predicate, to hide implementation details, to prevent auxiliary predicates from being called outside the object, and to take advantage of Logtalk advanced code encapsulating and reusing features.

Encapsulating Prolog code using Logtalk objects is simple. First, for each source file, add an opening object directive, `object/1`, to the beginning of the file and an ending object directive, `end_object/0`, to end of the file. Choose an object name that reflects the purpose of source file code (this is a good opportunity for code refactoring if necessary). Second, add public predicate directives, `public/1`, for the top-level predicates that are used directly by the user or called from other source files. Third, we need to be able to call from inside an object predicates defined in other source files/objects. The easiest solution, which has the advantage of not requiring any changes to the predicate definitions, is to use the `uses/2` directive. If your Prolog compiler supports cross-referencing tools, you may use them to help you make sure that all calls to predicates on other source files/objects are listed in the `uses/2` directives. In alternative, compiling the resulting objects with the Logtalk `unknown_predicates` and `portability` flags set to `warning` will help you identify calls to predicates defined on other converted source files and possible portability issues.

Prolog multifile predicates

Prolog *multifile* predicates are used when clauses for the same predicate are spread among several source files. When encapsulating plain Prolog code that uses multifile predicates, is often the case that the clauses of the multifile predicates get spread between different objects and categories but conversion is straight-forward. In the Logtalk object (or category) holding the multifile predicate *primary* declaration, add a predicate scope directive and a `multifile/1` directive. In all other objects (or categories) defining clauses for the multifile predicate, add a `multifile/1` directive and predicate clauses using the format:

```
:- multifile(Entity::Name/Arity).

Entity::Functor(...) :-
    ...
```

See the User Manual section on the `multifile/1` predicate directive for more information. An alternative solution is to simply keep the clauses for the multifile predicates as plain Prolog code and define, if necessary, a parametric object to encapsulate all predicates working with the multifile predicate clauses. For example, assume the following `multifile/1` directive:

```
% city(Name, District, Population, Neighbors)
:- multifile(city/4).
```

We can define a parametric object with `city/4` as its identifier:

```
:- object(city(_Name, _District, _Population, _Neighbors)).

    % predicates for working with city/4 clauses

:- end_object.
```

This solution is preferred when the multifile predicates are used to represent large tables of data. See the section on parametric objects for more details.

Converting Prolog modules into objects

Converting Prolog modules into objects may allow an application to run on a wider range of Prolog compilers, overcoming compatibility problems. Some Prolog compilers don't support a module system. Among those Prolog compilers which support a module system, the lack of standardization leads to several issues, specially with semantics, operators, and meta-predicates. In addition, the conversion allows you to take advantage of Logtalk more powerful abstraction and reuse mechanisms such as separation between interface from implementation, inheritance, parametric objects, and categories.

Converting a Prolog module into an object is easy as long as the directives used in the module are supported by Logtalk (see below). Assuming that this is the case, apply the following steps:

1. Convert the module `module/1` directive into an opening object directive, `object/1`, using the module name as the object name. For `module/2` directives apply the same conversion and convert the list of exported predicates into Logtalk `public/1` predicate directives.
2. Add a closing object directive, `end_object/0`, at the end of the module code.

3 Convert any `export/1` directives into `public/1` predicate directives.

4 Convert any `use_module/1` directives into `use_module/2` directives (see next section).

5 Convert any `use_module/2` directives referencing other modules also being converted to objects into Logtalk `uses/2` directives. If the referenced modules are not being converted into objects, simply keep the `use_module/2` directives unchanged.

6 Convert any `meta_predicate/1` directives into Logtalk `meta-predicate/1` directives by replacing the module meta-argument indicator, `:`, into the Logtalk meta-predicate indicator, `0`. Closures must be represented using an integer denoting the number of additional arguments that will be appended to construct a goal. Arguments which are not meta-arguments are represented by the `*` character.

7 Convert any explicit qualified calls to module predicates to messages by replacing the `:/2` operator with the `::/2` message sending operator, assuming that the referenced modules are also being converted into objects. Calls in the pseudo-module `user` can simply be encapsulated using the `{}/1` Logtalk external call control construct. You can also use instead a `uses/2` directive where the first argument would be the atom `user` and the second argument a list of all external predicates. This alternative have the advantage of not requiring changes to the code making the predicate calls.

8 If your module uses the database built-in predicates to implement module local mutable state using dynamic predicates, add both `private/1` and `dynamic/1` directives for each dynamic predicate.

9 If your module declares or defines clauses for multifile module predicates, replace the `:/2` functor by `::/2` in the `multifile/1` directives and in the clause heads (assuming that all modules defining the multifile predicates are converted into objects; if that is not the case, just keep the `multifile/1` directives and the clause heads as-is).

10 Compile the resulting objects with the Logtalk `unknown_predicates`, and `portability` flags set to `warning` to help you locate possible issues and calls to proprietary Prolog built-in predicates and to predicates defined on other converted modules. In order to improve code portability, check the Logtalk library for possible alternatives to the use of proprietary Prolog built-in predicates.

Before converting your modules to objects, you may try to compile them first as objects (using the `logtalk_compile/1-2` Logtalk built-in predicates) to help identify any issues that must be dealt with when doing the conversion to objects. Note that Logtalk supports compiling Prolog files as Logtalk source code without requiring changes to the file name extensions.

Compiling Prolog modules as objects

An alternative to convert Prolog modules into objects is to just compile the Prolog source files using the `logtalk_load/1-2` and `logtalk_compile/1-2` predicates (set the Logtalk `portability` flag set to `warning` to help you catch any unnoticed cross-module predicate calls). This allows you to reuse existing module code as objects. This has the advantage of requiring little if any code changes. There are, however, some limitations that you must be aware. These limitations are a consequence of the lack of standardization of Prolog module systems.

Supported module directives

Currently, Logtalk supports the following module directives:

`module/1`
: The module name becomes the object name.

`module/2`
: The module name becomes the object name. The exported predicates become public object predicates. The exported grammar rule non-terminals become public grammar rule non-terminals. The exported operators become public object operators but are not active elsewhere when loading the code.

`use_module/2`
: This directive is compiled as a Logtalk `uses/2` directive in order to ensure correct compilation of the module predicate clauses. The first argument of this directive must be the module **name** (an atom), not a module file specification (the adapter files attempt to use the Prolog dialect level term-expansion mechanism to find the module name from the module file specification). Note that the module is not automatically loaded by Logtalk (as it would be when compiling the directive using Prolog instead of Logtalk; the programmer may also want the specified module to be compiled as an object). The second argument must be a predicate indicator (`Name/Arity`), a grammar rule non-terminal indicator (`Name//Arity`), a operator declaration, or a list of predicate indicators, grammar rule non-terminal indicators, and operator declarations.

`export/1`
: Exported predicates are compiled as public object predicates. The argument must be a predicate indicator (`Name/Arity`), a grammar rule non-terminal indicator (`Name//Arity`), a operator declaration, or a list of predicate indicators, grammar rule non-terminal indicators, and operator declarations.

`reexport/2`
: Reexported predicates are compiled as public object predicates. The first argument is the module name. The second argument must be a predicate indicator (`Name/Arity`), a grammar rule non-terminal indicator (`Name//Arity`), a operator declaration, or a list of predicate indicators, grammar rule non-terminal indicators, and operator declarations.

`meta_predicate/1`
: Module meta-predicates become object meta-predicates. Only predicate arguments marked as goals or closures (using an integer) are interpreted as meta-arguments. In addition, Prolog module meta-predicates and Logtalk meta-predicates don't share the same explicit-qualification calling semantics: in Logtalk, meta-arguments are always called in the context of the *sender*.

A common issue when compiling modules as objects is the use of the atoms `dynamic`, `discontiguous`, and `multifile` as operators in directives. For better portability avoid this usage. For example, write:

```
:- dynamic([foo/1, bar/2]).
```

instead of:

```
:- dynamic foo/1, bar/2.
```

Another common issue is missing `meta_predicate/1`, `dynamic/1`, `discontiguous/1`, and `multifile/1` predicate directives. Logtalk allows detection of most missing directives (by setting its `missing_directives` flag to `warning`).

When compiling modules as objects, you probably don't need event support turned on. You may use the compiler flag `events(deny)` with the Logtalk compiling and loading built-in methods for a small performance gain for the compiled code.

Current limitations and workarounds

The `reexport/1` and `use_module/1` directives are not directly supported by the Logtalk compiler. But most Prolog adapter files provide support for compiling these directives using Logtalk's first stage of its term-expansion mechanism. Nevertheless, these directives can be converted, respectively, into `reexport/2` and `use_module/2` directives by finding which predicates exported by the specified modules are reexported or imported into the module containing the directive.

Finding the names of the imported predicates that are actually used is easy. First, comment out the `use_module/1` directives and compile the file (making sure that the compiler flag `unknown_predicates` is set to `warning`). Logtalk will print a warning with a list of predicates that are called but never defined. Second, use these list to replace the `reexport/1` and `use_module/1` directives by, respectively, `reexport/2` and `use_module/2` directives. You should then be able to compile the modified Prolog module as an object.

Although Logtalk supports term and goal expansion mechanisms, the semantics are different from similar mechanisms found in some Prolog compilers. In particular, Logtalk does not support defining term and goal expansions clauses in a source file for expanding the source file itself. Logtalk forces a clean separation between expansions clauses and the source files that will be subject to source-to-source expansions by using *hook objects*.

Dealing with proprietary Prolog directives and predicates

Most Prolog compilers define proprietary, non-standard, directives and predicates that may be used in both plain code and module code. Non-standard Prolog built-in predicates are usually not problematic, as Logtalk is usually able to identify and compile them correctly (but see the notes on built-in meta-predicates for possible caveats). However, Logtalk will generate compilation errors on source files containing proprietary directives unless you first specify how the directives should be handled. Several actions are possible on a per-directive basis: ignoring the directive (i.e. do not copy the directive, although a goal can be proved as a consequence), rewriting and copy the directive to the generated Prolog files, or rewriting and recompiling the resulting directive. To specify these actions, the adapter files contain clauses for the `'$lgt_prolog_term_expansion'/2` predicate. For example, assume that a given Prolog compiler defines a `comment/2` directive for predicates using the format:

```
:- comment(foo/2, "Brief description of the predicate").
```

We can rewrite this predicate into a Logtalk `info/2` directive by defining a suitable clause for the `'$lgt_prolog_term_expansion'/2` predicate:

```
'$lgt_prolog_term_expansion'(comment(F/A, String), info(F/A, [comment is Atom])) :-
    atom_codes(Atom, String).
```

This Logtalk feature can be used to allow compilation of legacy Prolog code without the need of changing the sources. When used, is advisable to set the `portability/1` compiler flag to `warning` in order to more easily identify source files that are likely non-portable across Prolog compilers.

A second example, where a proprietary Prolog directive is discarded after triggering a side-effect:

```
'$lgt_prolog_term_expansion'(load_foreign_files(Files,Libs,InitRoutine), []) :-
    load_foreign_files(Files,Libs,InitRoutine).
```

In this case, although the directive is not copied to the generated Prolog file, the foreign library files are loaded as a side-effect of the Logtalk compiler calling the `'$lgt_prolog_term_expansion'/2` hook predicate.

Calling Prolog module predicates

Prolog module predicates can be called from within objects or categories by simply using explicit module qualification, i.e. by writing `Module:Goal` or `Goal@Module` (depending on the module system). Logtalk also supports the use of `use_module/2` directives in object and categories (with the restriction that the first argument of the directive must be the actual module name and not the module file name or the module file path). In this case, these directives are parsed in a

similar way to Logtalk `uses/2` directives, with calls to the specified module predicates being automatically translated to `Module:Goal` calls. For example, assume a `clpfd` Prolog module implementing a finite domain constraint solver. You could write:

```
:- object(puzzle).

    :- public(puzzle/1).

    :- use_module(clpfd, [
        all_different/1, ins/2, label/1, (#=)/2, (#\=)/2,
        op(700, xfx, #=), op(700, xfx, #\=)
    ]).

    puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars ins 0..9,
        all_different(Vars),
                  S*1000 + E*100 + N*10 + D +
                  M*1000 + O*100 + R*10 + E #=
        M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0,
        label([M,O,N,E,Y]).

:- end_object.
```

As a general rule, the Prolog modules should be loaded (e.g. in the auxiliary Logtalk loader files) *before* compiling objects that make use of module predicates. Moreover, the Logtalk compiler does not generate code for the automatic loading of modules referenced in `use_module/1-2` directives. This is a consequence of the lack of standardization of these directives, whose first argument can be a module name, a straight file name, or a file name using some kind of library notation, depending on the back-end Prolog compiler. Worse, modules are sometimes defined in files with names different from the module names requiring finding, opening, and reading the file in order to find the actual module name.

Logtalk supports the declaration of predicate aliases in `use_module/2` directives used within object and categories. For example, the ECLiPSe IC Constraint Solvers define a `::/2` variable domain operator that clashes with the Logtalk `::/2` message sending operator. We can solve the conflict by writing:

```
:- use_module(ic, [(::)/2 as ins/2]).
```

With this directive, calls to the `ins/2` predicate alias will be automatically compiled by Logtalk to calls to the `::/2` predicate in the `ic` module.

When calling Prolog module meta-predicates, the Logtalk compiler may need help to understand the corresponding meta-predicate template. Despite some recent progress in standardization of the syntax of `meta_predicate/1` directives and of the `meta_predicate/1` property returned by the `predicate_property/2` reflection predicate, portability is still a problem. Thus, Logtalk allows the original `meta_predicate/1` directive to be overridden with a local one that Logtalk can make sense of. Note that the Logtalk library provides implementations of common meta-predicates, which can be used in place of module meta-predicates.

Logtalk allows you to send a message to a module in order to call one of its predicates. This is usually not advised as it implies a performance penalty when compared to just using the `Module:Call` notation. Moreover, this works only if there is no object with the same name as the module you are targeting. This feature is necessary, however, in order to

properly support compilation of modules containing `use_module/2` directives as objects. If the modules specified in the `use_module/2` directives are not compiled as objects but are instead loaded as-is by Prolog, the exported predicates would need to be called using the `Module:Call` notation but the converted module will be calling them through message sending. Thus, this feature ensures that, on a module compiled as an object, any predicate calling other module predicates will work as expected either these other modules are loaded as-is or also compiled as objects.

Compiling Prolog module multifile predicates

Some Prolog module libraries, e.g. constraint packages, expect clauses for some library predicates to be defined in other modules. This is accomplished by declaring the library predicate *multifile* and by explicitly prefixing predicate clause heads with the library module identifier. For example:

```
:- multifile(clpfd:run_propagator/2).
clpfd:run_propagator(..., ...) :-
    ...
```

Logtalk supports the compilation of such clauses within objects and categories. While the clause head is compiled as-is, the clause body is compiled in the same way as a regular object or category predicate, thus allowing calls to local object or category predicates. For example:

```
:- object(...).

    :- multifile(clpfd:run_propagator/2).
    clpfd:run_propagator(..., ...) :-
        ...    % calls to local object predicates

:- end_object.
```

The Logtalk compiler will print a warning if the `multifile/1` directive is missing. These multifile predicates may also be declared dynamic using the same `Module:Name/Arity` notation.