Logtalk 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].

Support for component-based programming

Predicates can be encapsulated inside *categories* which can be imported by any object. 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. 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.

Smooth learning curve

Logtalk has a smooth learning curve, by adopting standard Prolog syntax (using a preprocessor) 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 pre-processor 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 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.

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 with static binding implementations.

Last updated on: October 26, 2005