Universidade da Beira Interior
Departamento de Informática

Logtalk

Design of an Object-Oriented Logic Programming Language

Paulo Jorge Lopes de Moura

Thesis submitted in candidature for the degree of
Doctor of Philosophy (Computer Science)

Tese submetida à Universidade da Beira Interior para obtenção do grau de Doutor em Engenharia Informática

Covilhã
2003

Thesis written under the supervision of

Dr. Abel João Padrão Gomes

Assistant Professor of the Department of Informatics of the
University of Beira Interior

Tese realizada sob a orientação do

Professor Doutor Abel João Padrão Gomes

*Professor Auxiliar do Departamento de Informática da
Universidade da Beira Interior*

"There are no answers, only choices."

Gibarian to Kelvin in "Solaris", Steven Soderbergh's movie
based on Stanislaw Lem novel of the same name

"Não existem respostas, somente escolhas."

*Gibarian para Kelvin, no filme "Solaris" realizado por Steven Soderbergh,
baseado no livro de Stanislaw Lem com o mesmo nome*

To my parents

Aos meus pais

Acknowledgements

This work has been characterized by changes on advisors and work places. It started at the Department of Mathematics of the University of Coimbra on January of 1998, in a very hostile working environment to anything related to computer science. In September of 1999, the author moved himself to the University of Beira Interior to be closer to its beloved one. By the end of Spring of 2000, work on Logtalk and on the writing of this thesis almost stalled for more than a year due to personal problems. There are simply more important things in life than a Ph.D. thesis. With the help of my family and friends, things got better and I resumed work on developing Logtalk and on the writing of this thesis.

The decision to write this thesis in English was motivated by both the wish to improve my writing skills in this language and to ensure that my work will be available to a much wider audience than if it was written in Portuguese. This proved more difficult than anticipated. I wish to express my gratitude to Abel Gomes, Ana Prata, José Camões Silva, and Susana Ferreira for helping me in correcting the most blatant mistakes. Any remaining errors are, of course, my own responsibility. In addition, I want to thank my university for allowing me to write my thesis in English, a decision that always generates some controversy.

Many thanks to Abel Gomes, my Ph.D. advisor, for its continued support and friendship.

Thanks to all my colleagues for all their feedback regarding this work, especially Simão de Sousa for reading and commenting much of the final draft of this thesis in record time.

Thanks to all Logtalk users that contributed with bug reports, requests for improvements, and support by writing applications that provided test beds for the Logtalk language.

A special thanks to my family and friends for their enduring support, which is also an apology for my absence during the final writing stage of this thesis.

Abstract

This thesis describes the design, implementation, and documentation of Logtalk, an object-oriented logic programming language. Logtalk is designed as an extension to the Prolog logic programming language providing encapsulation features based on object-oriented concepts. Logtalk major features include support for both prototypes and classes in the same application, integration of event-driven programming with object-oriented programming, and category-based code reusing.

Keywords: Logtalk, Prolog, Logic programming, Object-oriented programming, Event-driven programming

Resumo

A presente tese descreve o desenvolvimento, implementação e documentação de uma linguagem de programação em lógica orientada para objectos denominada Logtalk, construída como uma extensão à linguagem de programação em lógica Prolog, implementando mecanismos de encapsulamento basedos nos conceitos da programação orientada para objectos. As principais características da linguagem Logtalk são o suporte de protótipos e classes na mesma aplicação, a integração da programação dirigida por eventos com a programação orientada para objectos e a reutilização de código baseada em categorias.

Palavras-chave: Logtalk, Prolog, Programação em Lógica, Programação Orientada para Objectos, Programação Dirigida por Eventos

Extended abstract in Portuguese

Este resumo alargado em língua portuguesa descreve os objectivos desta tese, a metodologia de trabalho seguida, as principais conclusões, assim como o trabalho planeado para o futuro.

Objectivos

Esta tese descreve o desenvolvimento de uma linguagem de programação em lógica orientada para objectos, denominada por Logtalk. A linguagem Logtalk foi construída como uma extensão à linguagem de programação em lógica Prolog, na medida em que implementa mecanismos de encapsulamento e de reutilização de código basedos nos conceitos da programação orientada para objectos. Este trabalho tem um conjunto de objectivos que podem ser divididos em dois grupos: objectivos científicos e objectivos técnicos.

Objectivos científicos

O princiapal objectivo científico é a integração dos conceitos da programação em lógica, da programação orientada para objectos e da programação dirigida por eventos numa mesma linguagem de programação.

Integração da programação em lógica com a programação orientada para objectos

A integração da programação em lógica com a programação orientada para objectos, sob a forma de uma extensão à linguagem Prolog, visa dotar esta linguagem de mecanismos de encapsulamento e reutilização de código baseados no conceito de objecto. Desta forma, o conhecimento sobre cada objecto do domínio de uma aplicação pode ser representado de forma declarativa. Para além disso, esta integração vem dar resposta a alguns dos problemas de engenharia de software que afectam a programação em Prolog quando aplicada a problemas de grande complexidade e dimensão.

Integração da programação dirigida por eventos com a programação orientada para objectos

A programação dirigida por eventos permite a construção de sistemas reactivos, caracterizados pelo facto de as computações serem iniciadas como consequência da ocorrência de eventos. Assim sendo, a programação dirigida por eventos complementa a programação orientada para objectos, na qual todas as computações resultam do envio de mensagens

a objectos. A ideia base para a integração destas duas formas de programação é estabelecer uma correspondência entre o conceito de evento e o envio de uma mensagem a um objecto. O programador define então que eventos observar e quais os objectos que actuarão como monitores desses objectos. Esta integração é fundamental na programação de relações entre objectos nas quais existem dependências entre o estado dos objectos participantes. A utilização de eventos permite minimizar as dependências entre os objectos e maximizar a sua coesão interna. A implementação dos conceitos de evento e monitor aumenta as capacidades de computação reflexiva da linguagem Logtalk, permitindo a definição fácil de ferramentas como, por exemplo, depuradores.

Suporte para sistemas baseados em protótipos e sistemas baseados em classes

Praticamente todas as linguagens de programação orientadas para objectos são baseadas em protótipos ou baseadas em classes. A grande maioria das linguagens é baseada em classes. No contexto da linguagem Prolog, a utilização de protótipos é a solução mais adequada para substituir a utilização de módulos. Tal como um módulo, um protótipo pode ser definido sem recursos a outras entidades e, ao contrário de classes e instâncias, não necessita de estar integrado numa hierarquia. No entanto, dependendo das aplicações, uma solução utilizando classes e instâncias poderá ser mais adequada do que uma solução baseada em protótipos. Por estes motivos, pretende-se integrar em Logtalk suporte quer para sistemas baseados em protótipos, quer para sistemas baseados em classes. A ideia é que este suporte seja implementado de forma a não priveligiar nenhum tipo de sistema e de forma a partilhar a generalidade das primitivas da linguagem, incluindo os mecanismos de envio de mensagens, os predicados pré-definidos sobre objectos e os métodos pré-definidos.

Objectivos técnicos

Para além dos objectivos científicos acima enunciados, existe também um conjunto de objectivos técnicos que visam dotar a linguagem Logtalk de paridade, a nível de características, com outras extensões ao Prolog e com outras linguagens de programação orientadas para objectos.

Suporte para múltiplas hierarquias de objectos

Um das características necessárias da linguagem Logtalk é o suporte para múltiplas hierarquias de objectos, uma necessidade que decorre da integração de protótipos e classes na mesma linguagem. Podemos assim ter várias hierarquias independentes de objectos, quer de protótipos, quer de classes.

Separação entre interface e implementação

A separação entre interface e implementação é uma característica fundamental de qualquer linguagem actual de programação. Esta característica não é contudo suportada pela maioria dos sistemas de módulos para Prolog, pela norma da ISO para módulos em Prolog, e pela maioria das extensões ao Prolog para programação orientada para objectos. No caso da linguagem Logtalk, pretende-se que uma mesma interface possa ser implementada por vários objectos (protótipos e classes) e que um mesmo objecto possa implementar várias interfaces.

Predicados privados, protegidos e públicos

Pretende-se que a linguagem Logtalk suporte a declaração de predicados privados, protegidos e públicos, com regras semelhantes às de outras linguagens de programação. Note-se que esta é uma característica que não é suportada pela norma corrente da ISO para módulos em Prolog. Nesta norma, como aliás na maioria dos sistemas de módulos disponíveis em compiladores Prolog, qualquer predicado encapsulado pode ser chamado desde que o seu nome seja conhecido.

Herança privada, protegida e pública

Pretende-se que a linguagem Logtalk suporte herança privada, protegida e pública, complementando o suporte para predicados de objectos privados, protegidos e públicos, de forma semelhante a outras linguagens de programação orientadas para objectos.

Objectos parametrizáveis

Um objecto parametrizável é um objecto cujo identificador é um termo composto contendo variáveis livres. Estas variáveis desempenham o papel de parâmetros, podendo ser utilizadas na definição dos predicados do objecto. Os objectos parametrizáveis podem ser vistos como uma forma de associarmos um conjunto de predicados a um termo composto (o identificador do objecto). Do ponto de vista da programação orientada para objectos, podemos usar parâmetros para representar o estado de um objecto que, sendo definido quando o objecto é criado, não é modificado durante o seu tempo de vida. O suporte para objectos parametrizáveis é uma característica comum a outras extensões ao Prolog.

Curva suave de aprendizagem

Pretende-se que a linguagem Logtalk seja desenhada e implementada como uma extensão natural à linguagem Prolog. O objectivo é conseguir que a linguagem Logtalk tenha uma curva suave de aprendizagem através da utilização de estruturas de controlo familiares aos programadores em Prolog e do recurso, sempre que possível, à sintaxe habitual do Prolog.

Compatibilidade com a maioria dos compiladores Prolog e com a norma da ISO

Um dos principais objectivos para a implementação da linguagem Logtalk é a compatibilidade com a generalidade dos compiladores Prolog e com a norma da ISO para esta linguagem. Este objectivo é tido em conta na concepção da linguagem de modo a serem minimizadas as características dependentes da implementação. Pretende-se assim maximizar a portabilidade quer da linguagem, quer dos seus programas.

Metodologia de trabalho adoptada

Os resultados obtidos neste trabalho são não só uma consequência dos objectivos acima enunciados, mas também da metodologia de trabalho adoptada. Esta segue os seguintes princípios:

- Implementar todas as características da linguagem.
- Testar a linguagem através de exemplos que tirem partido de cada uma das características da linguagem.
- Disponibilizar a implementação da linguagem no maior número possível de sistemas operativos e de compiladores Prolog.
- Recolher opiniões críticas sobre a linguagem através da disponibilização regular de versões públicas do compilador, da linguagem, dos exemplos e da documentação.
- Comparar soluções para problemas comuns em Logtalk com soluções implementadas através doutras linguagens de programação orientadas para objectos.
- Evitar descrever características não implementadas a menos que a sua viabilidade possa ser demonstrada esquematizando uma possível implementação.
- Ensinar a linguagem a estudantes de licenciatura que estejam familiarizados com outras linguagens de programação. Usar os resultados obtidos para melhorar a linguagem e a sua documentação.

Algumas das consequências destes princípios são:

- É necessário tomar uma decisão entre implementar um protótipo do sistema (como demonstração da sua viabilidade) ou implementar um sistema robusto e completo com todas as características da linguagem. Foi escolhida a segunda opção.
- A qualidade da documentação é essencial para conquistar utilizadores e programadores para a linguagem Logtalk.
- O desenvolvimento de bons exemplos é essencial para ajudar novos utilizadores a aprenderem a programar em Logtalk.
- Implementar, documentar e construir exemplos de programas que utilizem cada uma das características de uma linguagem permite demonstrar a simplicidade e viabilidade das nossas ideias.
- Disponibilizar versões públicas do Logtalk implica tomar decisões sobre que licenças de utilização adoptar e, consequentemente, sobre o suporte a disponibilizar para os utilizadores da linguagem.

Principais resultados

A linguagem Logtalk pode ser descrita como uma linguagem de programação multi-paradigma que suporta a programação em lógica, a programação orientada para objectos e a programação dirigida por eventos. No entanto, o objectivo da linguagem não é apenas suportar estes diferentes paradigmas mas também a sua integração. Esta integração foi conseguida, em primeiro lugar, pela reinterpretação dos conceitos de objectos no contexto da programação em lógica e, em segundo lugar, pela reinterpretação dos conceitos de eventos no contexo da programação orientada para objectos.

Logtalk como uma extensão ao Prolog

Na linguagem Logtalk o conceito de objecto é reinterpretado como contendo um conjunto de directivas de predicados (declarações) e um conjunto de cláusulas (definições). O conceito de envio de mensagem é reinterpretado como a construção de uma prova usando os predicados definidos para um objecto. Um método é simplesmente a definição de um predicado seleccionada para responder a uma mensagem. O conjunto de predicados definidos para um objecto é determinado através do uso de mecanismos de herança entre objectos. Ao reinterpretarmos os conceitos de objecto, mensagem e método nos termos da programação em lógica, estabelecemos um mapeamento simples entre a semântica da linguagem Logtalk e a semântica da linguagem Prolog.

De uma forma genérica, esta reinterpretação é comum à maioria das extensões ao Prolog para programação orientada para objectos. Para podermos comparar e diferenciar o Logtalk de outras extensões iremos utilizar os seguintes critérios: compatibilidade com compiladores Prolog, sintaxe, interpretação do conceito de objecto, características suportadas e ambiente de desenvolvimento.

Compatibilidade da implementação do Logtalk

O Logtalk é a única extensão ao Prolog para programação orientada para objectos, disponível actualmente, que foi projectada para ser compatível com a generalidade dos compiladores Prolog e com a norma da ISO para esta linguagem. Este objectivo diferencia o Logtalk das restantes extensões. A implementação da linguagem sob a forma de um pré-processador permite que a linguagem seja utilizada na generalidade dos sistemas operativos para os quais existe um compildor Prolog disponível. Concretamente, a versão corrente do Logtalk é compatível com trinta e uma versões de vinte compiladores Prolog.

Sintaxe do Logtalk

O Logtalk utiliza, sempre que possível, a sintaxe do Prolog, procurando definir primitivas que sejam elegantes e estejam de acordo com as expectativas dos programadores em Prolog, suavizando desta forma a curva de aprendizagem da linguagem. O que está aqui em causa é mais do que uma questão de açucar sintáctico. Por exemplo, o Logtalk permite encapsular e usar código Prolog sem que este necessite de sofrer quaisquer alterações. Apenas quando é necessário chamar predicados definidos noutros objectos é que necessitamos de fazer pequenas alterações ao código original. Isto assegura uma fácil reconversão de programas em Prolog.

O papel dos objectos na programação em lógica

O objectivo principal do suporte para objectos em Logtalk é o encapsulamento e a reutilização de código. Há assim uma separação entre esta funcionalidade e as questões teóricas das mudanças de estado dinâmicas que são características do conceito de objecto noutras linguagens. Desta forma, ao darmos privilégio às propriedades de encapsulamento e à reutilização de código dos objectos, a linguagem Logtalk pode ser utilizada como uma ferramenta efectiva para solucionar os problemas de engenharia de software que surgem quando usamos a linguagem Prolog em problemas de grande dimensão.

Soluções de implementação para conceitos de objectos

A linguagem Logtalk mostra como implementar em Prolog os principais conceitos da programação orientada para objectos. Uma parte destes conceitos não são encontrados individualmente na maioria das extensões existentes ao Prolog: suporte para protótipos e classes; meta-classes; protocolos e hierarquias de protocolos; predicados privados, protegidos e públicos; herança privada, protegida e pública. A linguagem Logtalk é assim uma das mais completas extensões ao Prolog para programação orientada para objectos. O Logtalk mostra também como implementar em Prolog outros conceitos importantes que não estão disponíveis noutras extensões como é o caso das categorias e da programação dirigida por eventos. Ao contrário de outras extensões que ou são proprietárias ou dependem fortemente dos detalhes de sistemas nativos de módulos, as soluções de implementação do Logtalk são compatíveis com qualquer compilador Prolog que siga, na generalidade, a norma da ISO para esta linguagem.

Objectos como alternativa aos módulos

Os objectos do Logtalk são uma alternativa à utilização de módulos na programação em Prolog. Tal como os módulos, podemos definir objectos como entidades autónomas de encapsulamento usando protótipos. Além disso, apesar do Logtalk não fornecer uma alternativa directa para as directivas de importação e exportação de predicados dos sistemas de módulos, as relações de extensão entre protótipos, juntamente com as relações de implementação de protocolos e as relações de importação de categorias permitem uma funcionalidade equivalente. Os objectos do Logtalk têm também importantes vantagens relativamente aos sistemas de módulos usados em compiladores Prolog e ao sistema de módulos especificado na norma da ISO:

- As directivas de visibilidade de predicados do Logtalk asseguram a protecção de predicados encapsulados, uma característica ausente na norma ISO para o sistema de módulos do Prolog.

Na linguagem Logtalk, os mecanismos de envio de mensagens e os predicados e métodos pré-definidos asseguram o cumprimento das directivas de visibilidade dos predicados. Em contraste, a norma ISO permite que qualquer predicado seja chamado através da qualificação explícita da chamada com o nome do módulo. Mecanismos de protecção de predicados são classificados como opcionais e dependentes da implementação da norma em cada compilador Prolog particular.

- Separação entre interface e implementação.

Ao contrário das interfaces de módulos, os protocolos do Logtalk podem ser implementados por qualquer número de objectos. Além disso, um objecto pode implementar qualquer número de protocolos.

- Compatibilidade com os compiladores Prolog existentes.

O Logtalk é compatível com praticamente todos os compiladores Prolog modernos. A norma ISO para módulos em Prolog não foi até ao momento adoptada pela generalidade dos compiladores, em boa parte devido às diferenças entre esta e os sistemas de módulos mais difundidos e utilizados.

- A norma da ISO especifica duas formas incompatíveis de declarar meta-predicados. Problemas deste género não existem em Logtalk.

Em todas as normas existem características que, pela sua natureza, não podem ser especificadas, ficando dependente das diferentes implementações. A sintaxe para declaração de meta-predicados não é certamente uma dessas caracterísiticas.

- O Logtalk suporta um conjunto de características importantes no contexto do desenvolvimento de projectos de grande dimensão, as quais estão fora do âmbito dos sistemas de módulos.

Estas características incluem a reutilização e especialização de predicados através dos mecanismos de composição e herança, a programação dirigida por eventos, a programação reflexiva e a geração automática de documentação de programas.

Ambiente de desenvolvimento e outras questões prácticas

O ambiente de desenvolvimento de programas em Logtalk é condicionado pelo subconjunto de características comuns aos compiladores Prolog compatíveis com a extensão. Por exemplo, não existe um conjunto comum de predicados para acesso às funcionalidades dos sistemas operativos, o que restringe a funcionalidade do compilador do Logtalk. Não existem também normas comuns para construir interfaces gráficas em Prolog. As extensões comerciais ao Prolog para programação orientada a objectos têm a vantagem de apenas terem de ser compatíveis com um único compilador, podendo assim tirar partido de características únicas, o que permite a construção de ambientes sofisticados de desenvolvimento. Apesar disso, e dentro das restrições de compatibilidade do Logtalk, a versão actual da linguagem é acompanhada de recursos que proporcionam uma funcionalidade semelhante aos ambientes de desenvolvimento não-gráficos de outras linguagens de programação. Por exemplo, o sistema contém ficheiros de configuração de editores de texto para reconhecimento da sintaxe dos programas em Logtalk. O sistema contém ainda um grande número de exemplos e documentação extensiva que inclui um manual do utilizador, um manual de referência e tutoriais.

Fora do mundo académico, estas questões práticas têm tanta importância como as características técnicas e os avanços científicos representados pela linguagem, sendo fundamentais na contrução de uma comunidade de utilizadores que utilize a linguagem Logtalk na resolução de problemas reais.

Logtalk como uma linguagem de programação orientada para objectos

O Logtalk estende a linguagem Prolog da mesma forma que, por exemplo, o CLOS estende a linguagem LISP ou o Objective-C estende a linguagem C. Enquanto linguagem de programação, o Logtalk partilha características com as linguagens mais comuns orientadas para objectos. Existem, no entanto, diferenças importantes devido ao facto do Logtalk ser baseado na linguagem Prolog. A diferença mais significativa é que o Logtalk elimina algumas das dicotomias enraizadas na maioria das linguagens orientadas para objectos. Estas dicotomias são frequentemente utilizadas para caracterizar e classificar estas linguagens. Especificamente, a linguagem Logtalk não faz distinção entre variáveis e métodos, suporta simultaneamente protótipos e classes e permite que as entidades e os seus predicados sejam estáticos ou dinâmicos.

Predicates como variáveis e métodos

Os predicados dos objectos Logtalk unificam os conceitos de métodos e variáveis de objectos, simplificando assim a semântica da linguagem. A utilização de predicados remove a dicotomia entre estado e comportamento: um predicado apenas declara aquilo que é verdadeiro sobre um objecto. Podemos usar predicados na representação quer de métodos, quer de variáveis, mas tal distinção é opcional. Daqui decorre que deixamos de necessitar de regras separadas para a definição, herança e visibilidade de estado e de comportamento. Desta forma podemos naturalmente partilhar estado e comportamento via herança ou, pelo contrário, definir ambos localmente num objecto. Podemos assim, por exemplo, definir métodos em instâncias e partilhar estado com os descendentes de um objecto sem que seja necessário formalizar conceitos como o de variáveis partilhadas de instância.

Elementos estáticos e dinâmicos da linguagem

Os objectos, protocolos, categorias e predicados do Logtalk podem ser estáticos ou dinâmicos. Além disso, podemos inserir e apagar cláusulas de predicados, quer em objectos estáticos, quer em objectos dinâmicos, durante a execucção. Podemos definir objectos, protocolos e categorias através de ficheiros com o código fonte ou criá-los dinamicamente durante a execucção. O programador não está limitado, por exemplo, a definir classes como entidades estáticas e instâncias como entidades dinâmicas que existem apenas durante a execucção. Podemos assim, por exemplo, definir uma instância num ficheiro como uma entidade estática, da mesma forma que podemos criar dinamicamente uma nova classe durante a execucção.

Suporte para protótipos e classes

O Logtalk é um linguagem *neutra* que suporta igualmente quer a programação baseada em protótipos, quer a programação baseada em classes. Ambos os dois tipos de objectos podem ser utilizados ao mesmo tempo, na mesma aplicação, trocando livremente mensagens entre si. Em Logtalk, classes, instâncias e protótipos são simplemente objectos — entidades de encapsulamento — caracterizados por diferentes conjuntos de regras sobre como aceder aos predicados neles encapsulados e nos seus antecessores. Classes e protótipos partilham os mesmos predicados pré-definidos para criar, apagar e enumerar objectos. Partilham também os mecanismos de envio de mensagens e os métodos pré-definidos para modificação dinâmica de predicados. Além disso, protocolos e categorias podem ser, respectivamente, implementados e importados por qualquer tipo de objecto.

A neutralidade da linguagem Logtalk traduz-se pela diversidade de tipos de sistemas de objectos que podem coexisitir numa mesma aplicação, todos eles suportados de forma igual. Podemos definir uma única hierarquia de classes (como em Smalltalk ou Java) ou múltiplas hierarquias de classes como em C++. Podemos também escolher entre implementar um sistema reflexivo definindo meta-classes ou usar as classes apenas como fábricas de instâncias. Como esperado, podemos também definir múltiplas hierarquias de protótipos. Tanto as hierarquias de classes como as hierarquias de protótipos podem usar apenas herança simples ou recorrer à herança múltipla. Esta flexibilidade na definição de sistemas de objectos é importante na utilização do Logtalk como ferramenta de aprendizagem da programação orientada para objectos.

Programação dirigida por eventos

A linguagem Logtalk integra a programação dirigida por eventos com a programação orientada para objectos. A chave para esta integração é interpretarmos o envio de uma mensagem como o único evento que pode ocorrer num programa. Podemos assim reinterpretar os conceitos de evento, monitor, notificação da ocorrência de um evento e método para processamento de um evento em termos de objectos, mensagens e métodos. Desta forma podemos escrever código usando os conceitos da programação dirigida por eventos sem abandonar o paradigma da programação orientada para objectos.

Da experiência de programação em Logtalk na implementação de relações complexas de dependências entre objectos emergem dois resultados importantes. Estes resultados não são específicos à linguagem Logtalk, aplicando-se a outras linguagens de programação orientada para objectos. Em primeiro lugar, a programação dirigida por eventos é uma característica essencial às linguagens de programação orientada para objectos para minimizar o acoplamento entre objectos e para maximizar a sua coesão interna em aplicações que implicam relações de dependências entre o estado dos objectos participantes. Em segundo lugar, os conceitos de evento e monitor devem ser implementados como primitivas das linguagens de programação. Este requisito é essencial em termos de desempenho, o que inviabiliza a implementação destes conceitos ao nível da aplicação. O suporte nativo para eventos e monitores é uma condição necessária para o uso efectivo da programação dirigida por eventos na resolução de problemas.

Composição baseada em categorias

As categorias são a base da programação baseada em componentes em Logtalk. Uma categoria permite o encapsulamento de unidades funcionalmente coesas de código que podem ser importadas por qualquer objecto. Uma categoria pode assim ser vista como um conceito dual do conceito de protocolo. O uso de categorias traduz-se por vários benifícios ao nível do desenvolvimento de programas: compilação incremental, actualização de um objecto — sem o recompilar — através de actualização das categorias importadas e o redesenho de objectos complexos através da sua separação em componentes.

A composição baseada em categorias é uma forma de reutilização de código que complementa a composição baseada em variáveis de objecto e herança. As categorias implementam uma forma de composição em que a interface da categoria passa a fazer parte da interface do objecto que a importa. Isto contrasta com o que acontece na composição baseada em variáveis de objecto, mas é similar ao que acontece na utilização de mecanismos de herança.

O conceito de categoria não depende de nenhuma característica específica e única da linguagem Logtalk. As categorias são compiladas utilizando as mesmas tecnologias usadas na compilação de objectos, com algumas funcionalidades a exigirem a utilização de ligação dinâmica entre mensagens e métodos. Podemos assim implementar o conceito de categorias noutras linguagens para as dotar de suporte para a programação baseada em componentes. O suporte em Logtalk para importação privada, protegida e pública de categorias estende a funcionalidade deste conceito. Através da aplicação dos princípios da programação baseada em componentes, o uso de categorias constitui uma alternativa de implementação para projectos que usam herança múltipla que pode ser aplicada no contexto de linguagens que apenas suportam herança simples.

Programação reflexiva

A linguagem Logtalk herda as caracterísiticas de meta-programação da linguagem Prolog, as quais representam uma forma de programação reflexiva. A estas caracterísiticas a linguagem Logtalk acrescenta o suporte para programação reflexiva estrutural e comportamental. A programação reflexiva é normalmente suportada através da programação das construções de uma linguagem usando a própria linguagem. No caso do Logtalk, a programação reflexiva é suportada por predicados pré-definidos, métodos pré-definidos e mecanismos de execução que não são programados em Logtalk. Isto implica que algumas das características da linguagem como, por exemplo, os mecanismos de envio de mensagens ou os mecanismos de herança, não podem ser redefinidos pelo programador. Esta limitação tem contudo como contrapartida a possibilidade de optimizarmos, em termos de desempenho, todas as características da linguagem que suportam a programação reflexiva.

O Logtalk suporta a programação reflexiva estrutural através de um conjunto de predicados pré-definidos que permitem a enumeração de entidades, das suas propriedades e das suas relações com outras entidades e através de um conjunto de métodos pré-definidos que permitem enumerar os predicados de um objecto e as proriedades de cada predicado. Estes predicados e métodos pré-definidos podem ser aplicados a protótipos, instâncias e classes.

O Logtalk suporta a programação reflexiva comportamental através da programação dirigida por eventos. Assim sendo, estamos restringidos a computações sobre as mensagens trocadas entre objectos. Apesar disso, a utilização de eventos permite a definição fácil de aplicações reflexivas como sejam, por exemplo, os depuradores.

A linguagem Logtalk suporta ainda a programação reflexiva através da definição de meta-classes. Cada classe pode ter a sua meta-classe como em Smalltalk ou partilhar uma meta-classe com outras classes. Podemos ainda definir meta-classes para todas as classes ou apenas para algumas. Desta forma a linguagem suporta praticamente qualquer tipo de sistema que use meta-classes.

Documentação de programas

A importância especial conferida em Logtalk à documentação de programas tem a sua origem nas ideias de *literate programming*. O suporte nativo em Logtalk para a documentação de programas difere da maioria das linguagens de programação em quatro pontos importantes:

- Os ficheiros que contém a documentação de uma entidade são automaticamente gerados quando esta é compilada.

O Logtalk usa uma única ferramenta — o compilador da linguagem — para compilar uma entidade e, em simultâneo, extrair a documentação. Com a ajuda de alguns programas escritos em linguagens de automação, exemplos dos quais são distribuídos com o compilador, reunir e pós-processar os ficheiros de documentação é uma tarefa simples.

- Os ficheiros de documentação são ficheiros XML válidos. Estes ficheiros contêm toda a informação sobre uma entidade que pode ter relevância em termos de documentação tal como, por exemplo, as relações da entidade com outras entidades e os predicados declarados pela entidade.

O formato XML permite representar informação de documentção sem haver necessidade de nos preocupar com a forma como essa informação irá ser formatada e apresentada. Um ficheiro XML pode ser facilmente convertido para formatos finais como o PDF (para imprimir) ou o HTML (para visualização no computador). É também possível processar os ficheiros XML para outros fins como, por exemplo, recolher dados sobre métricas de programas.

- A estrutura dos ficheiros de documentação faz parte da especificação da linguagem, juntamente com as directivas de documentação de entidades e predicados.

Por outras palavras, a documentação de programas é encarada como uma parte essencial da especificação da linguagem Logtalk.

- Toda a informação de documentação é expressa usando a linguagem Logtalk. Não existe uma linguagem adicional para documentação que necessite de ser dominada para podermos documentar programas.

As directivas de documentação do Logtalk podem ser estendidas pelo programador para representar informação arbitrária que não possa ser deduzida automaticamente a partir da compilação de uma entidade. Esta abordagem difere assim de outras linguagens de programação em que a documentação de programas recorre a comentários especialmente formatados.

Logtalk como ferramenta de ensino

O Logtalk foi utlizado na UBI (Universidade da Beira Interior) no ensino da programação orientada para objectos a alunos de licenciatura. Foi também utilizado como exemplo de uma extensão ao Prolog no ensino da programação em lógica. Esta experiência de ensino teve resultados interessantes. As linguagens mais comuns de programação orientada para objectos como o C++ ou o Java são baseadas em classes, as quais herdam sintaxe e conceitos de linguagens imperativas como o C. Estas linguagens requerem que conceitos como a alocação estática e dinâmica, a tipagem dos argumentos dos métodos, a importação de bibliotecas e outros sejam compreeendidos pelo aluno de modo a este poder escrever os programas mais simples. Estes conceitos são secundários e distraiem da aprendizagem de conceitos fundamentais como o encapsulamento, o envio de mensagens ou a herança. Em contraste, o Logtalk encapsula predicados, eliminando a necessidade de explicar os conceitos de métodos e variáveis antes de se definir simplesmente um predicado e enviar a respectiva mensagem. O suporte do Logtalk para protótipos e classes permite-nos podemos ensinar conceitos básicos usando simples hierarquias de protótipos antes de explicarmos a diferença entre classes e instâncias e entre relações de especialização e instanciação. Não temos palavras-chave como, por exemplo, `main`, `static`, `void`, `include` ou `import` a desviar a atenção do aluno de conceitos básicos ilustrados por simples exemplos. Não existe também um ambiente de desenvolvimento sofisticado como em Smalltalk, ao qual o aluno tem de se adaptar antes de poder escrever os exemplos mais elementares. Um editor de texto basta para escrever os primeiros programas em Logtalk. Para estudantes com conhecimentos básicos de programação em Prolog, a linguagem Logtalk é uma ferrementa ideal para uma transição suave da programação em lógica para a programação orientada para objectos. Tal deve-se quer ao uso de sintaxe e semântica derivadas do Prolog, quer devido à extensa gama de sistemas orientados para objectos suportados pelo Logtalk.

Números da distribuição da linguagem

Nos últimos dois anos o Logtalk foi descarregado da Internet uma média de 270 vezes por mês (9 vezes por dia). O sistema é ainda distribuído com o compilador Prolog YAP. As novas versões são em geral apenas anunciadas na lista de distribuição do Logtalk (cerca de 70 subscritores) e nas páginas do serviço Freshmeat (dedicado ao anúncio de software livre; cerca de 10 subscritores adicionais registados por esta via). Ponteiros para as páginas do Logtalk na Web figuram em cerca de uma centena de páginas de terceiros, que incluem desde páginas de recursos de programação até páginas pessoais de outros investigadores. Estes números da distribuição do Logtalk são modestos, mesmo se comparados com a dimensão da comunidade de programadores em Prolog. Eles mostram, no entanto, que existe um interesse neste tipo de extensões ao Prolog. Estes números revelam também que um maior esforço terá de ser feito na divulgação da linguagem.

Trabalho futuro

O desenvolvimento a curto prazo da linguagem Logtalk inclui tornar a linguagem mais atractiva para potenciais utilizadores, melhorando a documentação, a biblioteca de entidades e os exemplos que acompanham o compilador. Um dos objectivos principais é melhorar o suporte para o ensino dos conceitos da programação orientada para objectos aos alunos de licenciatura.

Os planos de desenvolvimento a médio prazo incluem a publicação dos resultados desta tese e o melhoramento da tecnologia utilizada no compilador actual do Logtalk. O trabalho desenvolvido deverá ser publicado em revistas técnicas e científicas que descrevam os principais resultados obtidos. Nomeadamente, a programação por componentes através do uso de categorias, o suporte simultâneo de classes e protótipos e a integração entre a programação orientada para objectos e a programação dirigida por eventos. No tocante ao desenvolvimento do compilador, o seguinte trabalho está planeado:

- O suporte de versões antigas de compiladores Prolog será abandonado, tornando a manutenção do Logtalk mais fácil e permitindo tirar partido de características apenas disponíveis em versões recentes de compiladores Prolog.
- O desempenho do envio de mensagens será optimizado através do uso, sempre que possível, de ligação estática entre mensagens e métodos.
- Uma nova implementação será desenvolvida modificando um compilador Prolog *open-source* de modo a obtermos uma versão autónoma do Logtalk. Esta versão contituirá uma implementação alternativa, complementando, mas não substituindo, a versão corrente da implementação do Logtalk através de um pré-processador.
- Um conjunto de testes será desenvolvido para ajudar a certificar a compatibilidade de compiladores Prolog com o Logtalk.

Um objectivo a longo prazo é conseguir que o Logtalk seja a norma *de facto* para a programação orientada para objectos em Prolog. Um dos principais obstáculos a ultrapassar é a percepção dos sistemas de módulos como solução suficiente para os problemas de engenharia de software em aplicações escritas em Prolog.

Preface

The first time I felt the need for strong encapsulation features in Prolog was during my final year undergraduate's project, in 1989. I was working on explanation-based learning and, as my work progressed, typical software engineering problems started to slow me down to a point where most changes would take weeks to implement. While seeking a solution, it became obvious that I needed a way to encapsulate code and define interfaces that would help me to deal with the increasing complexity of my application. Some Prolog compilers implemented module systems, but there wasn't one agreed standard. At the time, each compiler used its own implementation, a big handicap when writing portable code. Besides, my own computer science degree also lacked any object-oriented programming or software engineering classes.

I started to learn more about object-oriented programming soon after my graduation, in December 1989. After reading some papers on the design of Smalltalk, I decided that the best way for me to learn object-oriented programming was to build my own object-oriented language. As Prolog was always my favorite language, I decided to extend it with object-oriented features. Eventually, this work has led me to two generations of the Logtalk system. The Logtalk name is a tribute to Smalltalk, where the "Log" prefix refers to Logic.

The first Logtalk generation was developed while doing research work for my Master's thesis. I presented this work at a conference in September 1994 and released the first final public version in February 1995. The focus of this first system was to learn and experiment how to implement a class-based language. I also wanted to learn more about building reflexive systems. Although most users liked the ideas behind Logtalk, the lack of a preprocessor or compiler providing some level of syntactic sugar resulted in a system harder to learn and to use than necessary.

Another issue was the design choice of building a class-based system. I soon found that the concepts of class and instance were not a good alternative to the use of module systems for implementing libraries of predicates. Prototypes would have been a better option here, but my inexperience with prototype systems led me to postpone an implementation of this kind of system. At that time, it was also unclear to me how prototypes and classes could fit together in one language.

I also wanted to play around with and learn more about modern object-oriented programming language features, such as the separation of interface from implementation, parametric objects, and C++ like visibility rules. The big question was, as always, how these concepts would fit in a logic-programming context. In January of 1998, I started working on my Ph.D. program by designing and implementing a new system from scratch.

Based on feedback by early users and on subsequent work, the first development version of the second generation of Logtalk was released for registered users in July of

1998, followed by the first public beta in October 18, 1998. The final version went public on February 9, 1999 (the detailed history of Logtalk development can be found in the release notes distributed with the system).

I liked the Logtalk name, so I decided to keep it on the new system. This was a somewhat controversial decision among advisors and users. Many felt that, because I was building a new system from scratch, keeping the same name would lead people to believe that Logtalk 2.x was just the next version of Logtalk 1.x. Of course, this kind of decision is not uncommon in computer systems (the Mach micro-kernel comes to mind). Both systems share many ideas and goals but programs written for one system are not compatible with the other. However, conversion from the first system to the second can easily be accomplished in most cases.

The second generation of Logtalk was also a consequence of the desire to have a friendlier system, with a very smooth learning curve, bringing Logtalk programming closer to traditional Prolog programming. There were, of course, other important changes that resulted in a more powerful and pleasant system.

Logtalk 1.x releases occurred at a time where object-oriented programming was hyped everywhere, and that fact arguably contributed to its divulgation. By the time Logtalk 2.x came out, much of that hype had subsided, leaving many people disappointed by the once touted promises of the object-oriented technology. This reflected on the recent years drop of the number of researchers working in object-oriented logic programming systems, and in the number of Prolog object-oriented extensions being actively developed and supported. This is somehow ironic as Logtalk 2.x is a much better and mature system than any version of Logtalk 1.x ever was.

Convincing Prolog implementers and vendors to adopt an object-oriented extension to Prolog proved to be difficult. Most of them are willing to provide help in porting Logtalk, but, so far, the system is only distributed with the YAP compiler, only as a library, and not fully integrated with the compiler. Furthermore, the recent approval of the ISO standard for the Prolog module system also means that the official solution to Prolog software engineering problems is a technology that has been long deprecated in favor of object-oriented alternatives. Nevertheless, for some applications, modules are a good enough solution and, as such, they divert people from trying, learning, and developing more powerful and versatile solutions. It remains to be seen if people will favor ISO compliance or choose to work on alternative solutions like object-oriented subsystems. It does not make much sense to expect that a Prolog implementer will equally support two concurrent encapsulation mechanisms. However, even if a Prolog vendor embraces an object-oriented extension like Logtalk, module systems, will most likely, continue to be supported to ensure backward compatibility with older Prolog code. I hope that this thesis will be considered a worthy contribution for helping shaping the future of logic programming languages, and of Prolog in particular.

Paulo Moura
University of Beira Interior
May 2003

Contents

Introduction

Why developing Logtalk? The ultimate answer is, of course, for the fun of it. However, some readers might expect a different kind of answer. There are a number of reasons, consequence of the current state of affairs of both logic programming and object-oriented programming.

Regarding logic programming, Prolog, its most widespread language, born in 1972 [1], still lacks a standard library and a feature set suitable for programming in the large. The first Prolog ISO standard [2], concerning the core aspects of the language, was published in 1995, and is being adopted at a slow pace by the Prolog community. The second ISO standard [3], concerning the module system, was published in 2000 and is being basically ignored by most of the Prolog community. Long gone are the days of the Japanese 5th Generation Computer System Project [4], centered on the promises of logic programming as the silver bullet. Today, Prolog is a niche language. Logic was dropped as a programming paradigm from the first draft of the 2001 ACM/IEEE Computer Science Curricula [5], and was only re-introduced in the curricula final report [6] after much pressure from the logic programming community [7]. There are, of course, many reasons for the current state of logic programming in general, and Prolog in particular. Nevertheless, sociologic and political reasons aside, Prolog technical handicaps, such as the inexistence of standard libraries, standard foreign language interfaces, and powerful encapsulation mechanisms, make it an uphill battle to use the language and the logic programming paradigm for teaching, researching, or commercial software development.

Regarding object-oriented programming, the field is strongly biased towards class-based systems and the particular implementation of object-oriented concepts in a few languages, notably, C++ [8] and Java [9]. These languages, like any other language, represent a specific set of design decisions on how to implement broad object-oriented concepts. For example, these languages take for granted dichotomies such as variable/method, state/behavior, or instance/class. They present classes as static entities, and objects as runtime dynamic entities. Support for reflective computing is either minimal (C++ runtime type inference), or provided as a library, instead of being a core language feature (Java Reflection API). Not only other design decisions are possible, we can actually lift some of them to obtain a more general language by integrating object-oriented programming with logic programming. Unfortunately, the 2001 ACM/IEEE Computer Science Curricula cited above, most under-graduated courses, and many books, present class-based languages as almost synonymous of object-oriented programming. Prototype-based languages [10] are only a margin note, if that much, in most object-oriented programming courses and books. Innovative concepts pioneered in other languages and programming paradigms are yet to be found in mainstream object-oriented languages. One example is the concept of daemon, common in Artificial Intelligence languages. This concept is a cornerstone of the Logtalk support for

event-driven programming. Another example is the code's dual role of as both data and procedure found in logic and functional programming languages.

Goals

The major goal of this thesis work is the design, implementation, and documentation of an object-oriented logic programming language, named Logtalk, constructed as an object-oriented extension of the Prolog logic programming language. Its design aims to develop a language with a smooth learning curve for Prolog programmers, providing much needed powerful encapsulation features. Its implementation aims to demonstrate the feasibility of all language features, showing how they can be compiled in a clean and efficient way. Its documentation aims to provide a complete, clear, and detailed specification such that others can use it as a reference for providing alternative implementations. In this context, there are two sets of more specific goals: scientific and technical, which will be described next.

Scientific goals

The scientific goals of this thesis are the integration of the best features of logic programming, object-oriented programming, and event-driven programming paradigms in one language.

Integration of logic and object-oriented programming

Logtalk aims to bring together the main advantages of these two programming paradigms. On one hand, objects allow us to work with the same set of concepts in the successive phases of application development. On the other hand, logic programming allows us to represent, in a declarative way, our knowledge of each object. All together, objects and declarative programming allow us to shorten the distance between an application and its problem domain. Adding objects to Prolog allows us to apply high-level object-oriented development methodologies and metrics to logic programming. Objects also provide logic programming languages, and Prolog, specifically, with several features needed in large-scale software projects. In particular, objects add namespaces to the traditional Prolog flat database, providing predicate encapsulation and data hiding, enhance code reusing trough inheritance and composition, and, coupled with protocols, provide separation between interface and implementation.

Most Prolog object-oriented extensions available today are proprietary, commercial products. Almost all are class-based systems. Some of them are tailored for specific application areas. In contrast, Logtalk is an open-source project, aiming for broad compatibility with Prolog compilers, for supporting a wider range of object-oriented systems, and to be a general-purpose extension to Prolog.

Integration of event-driven and object-oriented programming

Event-driven programming enables the building of reactive systems, where computations are triggered by the observation of occurring events. This integration thus complements object-oriented programming, where computations are explicitly initiated by sending messages to objects. It should be accomplished by reinterpreting the event-driven programming concepts in terms of objects and messages and by an implementation that

minimizes any message sending performance penalties when events are not being used. The basic idea is to define an event as the sending of a message to an object. The user then dynamically defines which events are to be observed and sets up monitors for them. This enables clean and elegant solutions that minimize object coupling and maximize object cohesion. A good example is the representation of relations between objects that imply constraints on the state of participating objects [11, 12, 13, 14]. Another example are reflective applications such as code debugging and code profiling [15]. Event-driven programming is also at the core of graphical user interfaces and operating systems [16]. The notion of event-driven programming also has roots in the concept of daemon found in Artificial Intelligence languages [17, 18]. Some object-oriented languages already support some sort of event-driven programming. For example, Smalltalk [19] provides a limited form of this idea through its dependency mechanism, which is implemented at the class level. Java provides a similar mechanism at the API level. However, most widespread object-oriented languages used today, such as C++ or Java, include no *native* support for event-driven programming at the language level. Logtalk aims to demonstrate that event-driven programming not only can be nicely integrated in an object-oriented language, but also is essential in many applications to avoid breaking object encapsulation and to avoid rewriting object code when reusing objects in unforeseen applications.

Support for both prototype-based and class-based systems

Almost all object-oriented languages available today, are either class-based or prototype-based [20], with a strong predominance of class-based languages. In particular, all current mainstream object-oriented languages are class-based. However, prototypes provide a much better replacement for Prolog modules than classes. A prototype can be a stand-alone object, not attached to any hierarchy, and therefore a convenient solution to encapsulate code that will be reused by several unrelated objects. Prototypes are thus a natural upgrade to the use of modules in applications where the concepts of instantiation and specialization of class-based languages do not make sense. For other applications, the forms of code reuse underlying the concepts of class and instance, are the best solution. Each kind of system, with its strengths and drawbacks, is equally useful in the context of an object-oriented logic programming language. As such, Logtalk aims to provide equal support for classes and prototypes, including runtime support for both prototype and class hierarchies in the same application. In fact, many Logtalk examples and applications make use of prototypes, classes, and instances simultaneously.

Technical goals

In addition to the scientific goals stated in the previous section, there is also a set of technical goals intended to give Logtalk feature-parity with current object-oriented programming languages, to provide an easy migration path for Prolog programmers, and to ensure broad compatibility with today's Prolog compilers and ISO standards.

Support for multiple object hierarchies

Languages such as Smalltalk-80, Objective-C [21], and Java, define a single hierarchy with a root class usually named `Object`. This makes it easier to ensure that all objects share a common behavior. Unfortunately, this often results in lengthy hierarchies, where

most inherited behavior is never used by descendant objects [22]. Moreover, a single object hierarchy makes it difficult to express objects that are exceptional in some sense when compared to other objects. Therefore, one of the Logtalk design goals is to provide support for multiple, independent, object hierarchies. This goal is also a consequence of the Logtalk support for both prototypes and class hierarchies.

Separation between interface and implementation

This is an expected feature of any modern high-level programming language. Logtalk should provide support for separating interface from implementation in a flexible way, enabling an interface to be implemented by multiple objects, and an object to implement multiple interfaces. Surprisingly, this is not possible in the current ISO standard for Prolog modules, where a module consists of a single module interface and zero or more corresponding module bodies.

Private, protected, and public object predicates

Logtalk should support data hiding by implementing private, protected, and public object predicates, and by adopting scope rules common to other object-oriented languages: private predicates can only be called from the container object, protected predicates can only be called from the container object and its descendants, and public predicates can be called from any object. Note that the current ISO standard for Prolog modules does not support data hiding. By using explicit module qualification, we can call any module predicate as long as we know its name.

Private, protected, and public inheritance

This is a common feature of modern object-oriented languages such as C++ and Java, and a natural extension of the predicate scope rules. Logtalk should support a generalized implementation of private, protected, and public inheritance, enabling us to restrict the scope of inherited, implemented, or imported predicates.

Parametric objects

A parametric object is an object whose identifier is a compound term containing free variables. These variables play the role of object parameters. Object predicates can then be coded to depend on the parameter values. Object parameters can also be used to represent object state that is set at creation time and will not be modified during object lifetime, thus avoiding the use of destructive update methods for object initialization. Parametric objects can be seen as a way of associating a set of predicates (the object methods) with a compound term (the object identifier). Parametric objects are already implemented and proved a valuable feature in some object-oriented logic programming systems such as L&O [23] and SICStus Objects [24].

Smooth learning curve

Logtalk should be designed as a natural extension to Prolog, not a completely new, radically different approach to logic programming. Logtalk should strive for a smooth

learning curve by adopting, whenever possible, standard Prolog syntax, by using programming constructs familiar to Prolog users, and by allowing incremental learning and use of most of its features.

Compatibility with most Prolog compilers and the ISO standard

Logtalk should be designed to be compatible with most Prolog compilers and, specifically, with the ISO Prolog standard. It should run on most computer systems supporting a modern Prolog compiler. The language design should minimize implementation-dependent features to ensure broad portability of Logtalk programs across Prolog compilers and operating systems. In fact, Logtalk should be regarded as an ideal tool for writing portable programs: any operating-system dependent code can be encapsulated inside objects implementing clearly defined cross-platform protocols.

Work methodology

I felt compelled to describe my working methodology during the design of Logtalk. I believe that the results of this work are a consequence, not only of the stated goals, but also of the way I pursued those goals. While doing my research work, I tried to comply with the following guidelines:

- Implement all language features.
- Experiment by writing examples that take advantage of every provided feature.
- Port the language compiler to as many operating systems and Prolog compilers as possible.
- Collect user feedback by regularly releasing public versions of the language compiler, examples, and documentation.
- Compare Logtalk solutions for common and classical problems to solutions implemented in other object-oriented languages.
- Avoid writing about unimplemented features unless they can be proved feasible by sketching a possible implementation.
- Teach the language to undergraduate students already familiar with some programming languages. Use their feedback to improve the language and its documentation.

Some of the consequences of this methodology are:

- A decision must be made between implementing a prototype, proof-of-concept system, or implementing a feature-complete, robust system. The later option was chosen.
- Good documentation is essential to attract Logtalk users and developers.
- Good examples are essential to help new users learn how to write programs in Logtalk.

- Implementing, describing, and exemplifying language features provides hard evidence of the simplicity and feasibility of our ideas.
- Releasing public versions of the Logtalk system implies making decisions on what software licenses to adopt and, consequently, on what kind of support we will find ourselves obliged too.

Reader background

On this thesis, I assume that the reader have a working knowledge of both logic programming and object-oriented programming. I do no attempt to explain the basics of any of these programming paradigms. As in much multi-disciplinary work, most readers will feel more comfortable with one programming paradigm that with the other. I hope to convince them that much is to be gained by integrating these two programming paradigms into the same language.

Describing the design and development of the Logtalk language in the linear way imposed by a thesis is not an easy task. Therefore, sometimes I will need to refer to concepts that will only be fully explained in a later chapter. However, a basic knowledge of both logic programming and object-oriented programming should be enough to get the reader through this thesis.

Thesis outline

The remaining of this thesis is structured as follows:

Chapter 1 presents the Logtalk concept of object (including parametric objects), explains the language support for both class-based and prototype-based hierarchies, compares Logtalk to other object-oriented extensions and to ISO Prolog modules, explains the rules for protocol and implementation inheritance, and illustrates the use of objects through several examples.

Chapter 2 describes the control constructs defined in Logtalk for message sending, message broadcasting, predicate specialization, and for bypassing the pre-processor when compiling predicate definitions.

Chapter 3 explains how to declare, define, redefine, and specialize object predicates, describes the built-in predicates and built-in methods defined in Logtalk, and illustrates through examples how to implement common object-oriented concepts using object predicates.

Chapter 4 describes the Logtalk concept of protocol, compares Logtalk protocols with related concepts in other programming languages, explains how to define protocols and protocol hierarchies, and illustrates the use of protocols through several examples.

Chapter 5 explains the Logtalk concept of category (a first-class entity that encapsulates code that can be imported by any object), compares code reuse by using categories with code reuse by using inheritance and instance-based composition, and illustrates the use of categories through several examples.

Chapter 6 shows how to integrate event-driven programming with object-oriented programming, presents the Logtalk concepts of event and monitor, and describes the Logtalk language support for event-driven programming.

Chapter 7 describes the Logtalk language support for representing documenting information about entities and predicates and how to automatically extract and format such documentation.

Chapter 8 describes the implementation-specific Logtalk design goals, discusses the issues found while porting Logtalk to several Prolog compilers and operating systems, and provides a high-level description of the current Logtalk language implementation.

Conclusions summarizes the major contributions of this thesis and presents a roadmap for future development.

Appendix A formally defines the Logtalk language syntax using a derivated Backus-Naur Form notation.

Appendix B contains a complete description of all built-in predicates, built-in methods, directives, and control constructs of the Logtalk language.

Appendix C formally defines the format of the Logtalk automatically-generated entity documenting files and presents some examples on how to transform these files into printing and on-line viewing formats.

Some of the research work described in this thesis, such as the Logtalk language reference and the Logtalk grammar, has been previously published in a technical report [25]. The issues related to Logtalk porting, described in Chapter 8, have first been published in a technical paper [26]. Some of the ideas behind Logtalk were first published in a paper [27]. The current Logtalk implementation (that includes most of the programming examples presented on this thesis) is available from the Logtalk web site [28].

Chapter 1

Objects

This chapter begins by presenting the Logtalk concept of object, followed by the definition of common object-oriented terms in the context of the Logtalk language. Secondly, Logtalk is presented as a neutral language supporting several types of object-oriented systems, including class-based and prototype-based ones. Thirdly, the definition of objects and object hierarchies is explained, along with a description of built-in object directives, built-in predicates for object handling, and built-in object reflection predicates. Fourthly, some simple examples are presented, including class-based and prototype-based solutions for the same programming problem. Next, Logtalk support for parametric objects is described and illustrated through several examples. Finally, Logtalk is characterized as both a class-based and as a prototype-based language.

1.1 Logtalk object concept

The Logtalk goal of adding objects to Prolog is the encapsulation and reuse of predicates. As such, the issues of dynamic state in the context of logic programming are not a research topic of this thesis. In most object-oriented programming languages, an object is essentially a glorified dynamic data structure. From this point-of-view, objects provide encapsulation of dynamic state. However, we can also view (the primary purpose of) objects as a way of encapsulating and reusing code. In the case of Logtalk, an object encapsulates directives and predicate clauses. Dynamic object state is available using assert and retract built-in methods, but, as with dynamic predicates in Prolog programming, it should be only used when strictly necessary.

Logtalk objects work as namespaces for the encapsulated predicates. This is an important feature over the flat predicate database model of plain Prolog. Support for namespaces allows us to reuse predicate names, simplifying the vocabulary of our applications, and to partitioning our code in more manageable parts, essential in the construction of large programs. By itself, namespaces is a feature already available in Prolog using modules. However, as it will be demonstrated throughout this thesis, objects provide several advantages over modules for encapsulating and reusing code, while subsuming their functionality.

1.1.1 Object-oriented concepts

Designing a Prolog object-oriented extension implies defining the set of object concepts that will be implemented. This set includes both core object concepts and comple-

mentary and derivate concepts that depend on the design goals. Taking into account the goals expressed in the previous chapter, the core object concepts must include: support for both prototypes and classes, separation of interface from implementation, and data hiding through the declaration of public, protected, and private predicates. Complementary and derivate concepts include: multiple object hierarchies (partially a consequence of the support for both classes and prototypes), parametric objects, and public, protected, and private inheritance (complementing data hiding). The way these concepts are implemented depends on the intended uses of the object-oriented extension. In the case of Logtalk, the implementation of object-oriented concepts is guided by the view of objects as encapsulation units. Thus, Logtalk emphasis is on the encapsulation and reuse of predicates. Unlike other extensions, which are developed under different goals, Logtalk does not try to bring to Prolog concepts of attributes and methods, destructive assignment primitives, typed slots, or similar concepts as found on common object-oriented languages such as C++ or Java. A comparison between Logtalk and other object-oriented extensions is included later in this chapter.

1.1.2 Objects, classes, and prototypes

In Logtalk, the terms *object*, *prototype*, *parent*, *class*, *subclass*, *superclass*, *metaclass*, *instance*, and *ancestor* always designate an object. Different names are used to emphasize the role that an object plays in a particular context. The role an object plays depends on its relation with other objects. We use a term other than *object* when we want to make the relationship with other objects explicit. As will be discussed later in this chapter, prototypes, classes, and instances are characterized by different rulesets for accessing and inheriting predicates. There are only three kinds of entities in Logtalk: objects, protocols, and categories. Protocols encapsulate predicate declarations, enabling the separation between interface and implementation. Categories encapsulate both predicate declarations and definitions that can be imported by any object. Protocols and categories will be fully discussed in Chapters 4 and 5, respectively.

Definitions

We start by defining informally a set of terms useful for the object classification that will be used throughout this thesis:

object An encapsulation entity, characterized by an identity, a set of entity directives, a set of predicate directives, and a set of predicate clauses. The identity of an object is either an atom or a compound term. Logtalk objects can be either static or dynamic, similar to Prolog code.

parametric object An object whose identity is a compound term containing free variables, which can be used to parameterize the object predicates.

class An object that declares (and possibly defines) the predicates common to a set of objects (its instances).

abstract class A class that cannot be instantiated and that is used to encapsulate predicates that are inherited by other classes.

subclass A class that is a specialization (directly or indirectly) of another class.

superclass A class from which another class is (directly or indirectly) a specialization.

instance An object whose predicates are declared by its classes (and by the superclasses of its classes).

metaclass The class of a class, when it is interpreted as an object. Thus, metaclass instances are themselves classes. In a reflexive system, any metaclass is also an instance of some class.

prototype A self-describing object that declares (and possibly defines) its own predicates. A prototype may extend or be extended by other prototypes.

parent A prototype that is extended by another prototype, thus defining a prototype hierarchy.

ancestor A class or a parent declaring (and possibly defining) predicates that are used by a descendant object. In class-based hierarchies, the ancestors of an instance are its classes and their superclasses. In prototype-based hierarchies, the ancestors of a prototype are its parents and their ancestors.

1.1.3 Logtalk as a neutral, unbiased object-oriented language

A major goal of Logtalk is to support both classes and prototypes. The strengths and weaknesses of class-based and prototype-based languages has long been a research topic (see, for example, [29, 30, 31, 32, 33]). Today, most object-oriented languages, including the most widespread ones, are class-based. Prototype-based languages are a minority, both in terms of market share and of mind share. Nevertheless, prototype and classes provide some complementary features regarding code encapsulation and reuse. Both interpretations of objects are useful in the context of an object-oriented logic programming language.

Prototypes can be seen as self-describing objects. They can be defined as stand-alone objects, not attached to any hierarchy. There is no need to create a class and instantiate it in order to represent and use an object. Prototypes may be defined by stating what is different from other prototype, using an extension mechanism. Singular objects and exceptions are easily represented because what is different can be locally represented. We only have a single hierarchical relation, making prototype hierarchies easy to grasp. Unfortunately, it is not possible to encapsulate knowledge in an object only to be used by other objects, not by the object itself. When we send a message to a prototype, the lookup search starts in the object itself. If the object cannot answer the message, then it is delegated to the prototypes that it extends. We can adopt conventions to represent the equivalent of classes, but the very nature of prototype systems does not help to enforce them. Prototypes are the most natural alternative to the use of Prolog modules. For example, a module containing list-handling predicates can easily be replaced by a prototype: exported module predicates become public prototype predicates.

A class represents an abstraction of the common characteristics of a set of objects. As such, it enables easy representation of hierarchies of classification. A class enables the encapsulation of knowledge to be reused by other objects, but the class itself as an object. This can be either an advantage or a problem, depending on what we are trying to represent. One of its drawbacks is the so-called "single-instance class" problem: representing a unique object implies the definition of a class that has only one instance.

This is cumbersome to implement. Classes can play the role of instance factories, handy when we need to create several similar objects. Assume, for example, that we want to represent the state space of classical problems such as the "missionaries and cannibals" puzzle. A state space can be abstracted in a class that declares predicates for representing the initial state, the successors of a state, and the goal state. Class instances will represent concrete problems by defining these predicates. In addition, the class representing state spaces can be easily specialized by a class representing heuristic state spaces.

From the point-of-view of Logtalk, the principal difference between classes and prototypes is that their implied relations with other objects result in different forms of reuse of the encapsulated predicates. In class-based languages, a distinction is made between abstractions and concrete objects, resulting in two kinds of object relations: specialization relations between classes, and instantiation relations between classes and concrete objects. In prototype-based languages, there is only one relation between objects, the extension relation. The practical consequences of using either a class-based solution or a prototype-based one will be exemplified later in this chapter.

Compiling objects

Supporting both classes and prototypes in the same language raises some questions with respect to object compilation, mainly when to compile an object as a prototype and when to compile an object as a class or as an instance. If an object extends another object, it is compiled as a prototype. On the other hand, if an object specializes or instantiates other objects, it is compiled as an element of class-based hierarchy. In the case of an object that is not hierarchically related to any other object, it is always compiled as a prototype. However, there is a problem when we try to define a class hierarchy similar to the one in Java. In a non-reflective class-based hierarchy, the root class does not instantiate or specialize any other class. Nevertheless, it must be compiled as a class and not as a prototype. The solution is to use always a minimal reflective class-based design, as illustrated later in this chapter.

1.2 Related work

In this section, comparisons are made between Logtalk and other Prolog object-oriented extensions, and with Prolog module systems, including the ISO Prolog standard for modules. Comparisons with other object-oriented languages are made through this thesis whenever necessary.

1.2.1 Prolog object-oriented extensions

A comprehensive description of the most common Prolog object-oriented extensions can be found in [34]. In recent years, some new extensions have been released. In this section, a brief description of the most important features of current Prolog object-oriented extensions is presented. Only Prolog object-oriented extensions that are being actively maintained are covered here.

Quintus Objects

Quintus Objects [35] is a class-based object-oriented extension for the Quintus Prolog [36] commercial compiler. In Quintus Objects, an object is an updatable data structure whose slots can be declared as private or protected. The type of a slot can be either a Prolog term or C-style type. Slots can be directly accessed without using message sending. Getter and setter methods for slots are automatically created when objects are compiled. Single and multiple inheritance are supported using static binding. As most extensions, Quintus Objects works as a preprocessor, translating objects to Prolog code at compile-time. Debugging is supported through a directive that enables tracing of erroneous use of message sending.

SICStus Objects

SICStus Objects [37] is a prototype-based object-oriented extension that is included with SICStus Prolog [24], a commercial compiler. It is implemented using modules. Object predicates are public, so SICStus Objects does not provide data hiding. This extension supports a concept of attribute, providing two built-in methods that enable efficient access and definition of attribute values. Both message sending and message delegation are supported. Although SICStus Objects is prototype-based, it also supports a concept of instance. An instance is a light weight, efficient object that is initialized with a copy of the attributes of a prototype, which works as its class. SICStus Objects supports parametric objects.

Jinni

Jinni [38, 39] is a Java-based commercial Prolog system. It includes a class-based object-oriented extension. It features multiple inheritance (using an algorithm that prevents cycles), static binding, and the definition of constructors that are called at instance creation-time (and automatically call the constructors of the superclasses). Jinni does not support data hiding in its current version (i.e. all object predicates can be called). It defines concepts of instance fields and class fields (shared instance fields) with predefined setter and getter operations. An interesting feature is that classes may be located at arbitrary locations on the Internet.

OPL

OPL [40] is a commercial class-based object-oriented programming extension for the Amzi! Prolog compiler [41]. It supports multiple inheritance and dynamic reclassification. It makes a distinction between services (methods) and attributes (whose names must be atoms), defining built-in operations for setting, accessing, and deleting its values. Attributes can have multiple values. Class attributes work as shared instance attributes. OPL defines a single metaclass for all classes. This metaclass defines a set of general services, including debugging through tracing of message sending, application initialization, and saving and restoring the state of an application to a disk file.

O'CIAO

O'CIAO [42, 43] is a set of libraries for class-based object-oriented programming in CIAO Prolog [44, 45]. Both the extension and the Prolog compiler are open source projects.

O'CIAO object-oriented programming model is based on the CIAO Prolog module system. Classes are a special type of modules that can be instantiated. The syntax of O'CIAO uses a mixture of object-oriented and module terminology. For example, public predicates are declared using `export/1` directives, but protected predicates are declared through `inheritable/1` directives (predicates are private by default). Predicate scope declarations must be repeated in the subclasses. A class is required to provide an implementation for every declared public predicate. In addition, in order to allow a predicate to be redefined in subclasses, we must explicitly use a `virtual/1` predicate directive. Constructor and destructors predicates can be explicitly declared; they are automatically called when an instance is created and deleted, respectively. O'CIAO makes a distinction between predicates that are attributes and predicates that represent methods. Attributes are used to represent the internal state of instances; they are restricted to Prolog facts. Methods are static predicates defined in classes. O'CIAO supports the definition of interfaces. These are similar to Java interfaces. In respect to inheritance, O'CIAO supports single inheritance of implementation and multiple inheritance of interfaces.

Minerva

Minerva [46] is a commercial Java-based Prolog implementation, tailored for web applications. Its latest version (Minerva 2.4) introduces a new class-based extension. This extension is oriented to the encapsulation of dynamic state. Object creation methods are automatically created when a class is compiled. Object state handling is performed using blackboard-like built-in predicates where the first argument is the instance name. There is no support for inheritance or subclassing.

LPA Prolog++

Prolog++ [47, 48] is a commercial product of LPA [49]. It is a class-based system, supporting single and multiple inheritance. Prolog++ makes a distinction between methods and attributes. Methods are static, while attributes are dynamic. Accessing and assignment primitives are defined to handle attributes. Both methods and attributes can be either public or private. Prolog++ includes several graphical development tools such as an object browser, an editor for object methods, and an object-graph to handle class hierarchies. Two interesting features of Prolog++ are its support for part-of hierarchies and data-driven programming.

ISCO

ISCO [50] is a logic programming language developed for interfacing with organizational information systems such as databases. It allows information to be described using single-inheritance hierarchies of classes. ISCO is currently implemented as a Prolog program and a set of front-ends for Web interfacing and for accessing databases. ISCO is one of several examples of logic programming languages adopting object-oriented concepts to better deal with the complexity of programs that work on heterogeneous environments.

XPCE

XPCE [51] is a class-based object-oriented system for building graphical user interfaces. It includes a rich library of built-in classes implementing graphics widgets and data structures, organized in a single-inheritance hierarchy. It is written in C, but allows object methods to be defined in multiple languages, including Prolog. XPCE works as a library for a host language such as Prolog, Lisp, or C++. Thus, XPCE is not a Prolog object-oriented extension in the same sense as the ones described above, but can be used as such. In its current version, it is only compatible with the SWI-Prolog compiler, although older versions worked with both Quintus Prolog and SICStus Prolog.

Other Prolog object-oriented extensions

Two other Prolog object-oriented extensions, L&O [23] and OL(P) [52], should be mentioned here. They are no longer in development. However, the source code of both extensions is publicly available and has inspired some of the implementation solutions adopted by Logtalk (and other Prolog object-oriented extensions). L&O is compatible with both IC-Prolog and LPA MacProlog, while OL(P) is compatible with Quintus Prolog and SICStus Prolog. As mentioned above, a summary of the major features of both extensions can be found in [34].

1.2.2 Why developing Logtalk?

From the strict point-of-view of a Prolog object-oriented extension, why developing Logtalk? Assuming that the goals expressed in the previous chapter have been reached, Logtalk provides a set of features that, taken altogether, are not available in any other Prolog extension. However, Logtalk also represents a set of design choices, and consequently a set of trade-offs, that differentiates it from most other Prolog object-oriented extensions. Some of these differences are discussed below. In addition, it should be noted that Logtalk is one of the few extensions under active development that is not a commercial product.

Compatibility with Prolog compilers

The compatibility with the generality of Prolog compilers of most extensions is very limited. Two older extensions, OL(P) and L&O, could be ported to other Prolog compilers, but they are no longer being maintained or developed. Most of the remaining extensions are proprietary, closed products. This is the case of Quintus Objects, Jinni, OLP, Prolog++, and Minerva. Two other extensions, SICStus Objects and O'CIAO, rely on term expansion mechanisms for source-to-source compilation that are only available in some Prolog compilers. In addition, their implementations depend heavily on the details of the native module systems. This constitutes another roadblock to porting these extensions to other Prolog compilers. In contrast, Logtalk is an open source project, designed from scratch to be compatible with most Prolog compilers and with the current ISO Prolog standard. The Logtalk implementation does not depend on details specific to any Prolog compiler.

Areas of application

Some object-oriented extensions are tailored for specific areas of application. For example, XPCE is mainly used for developing graphical user interfaces. Quintus Objects is optimized for writing efficient object-oriented programs in Prolog where objects are essentially dynamic data structures containing Prolog terms and C-like typed values. ISCO is geared towards the development and maintenance of heterogeneous information systems.

The scope of application of most extensions described above is also delimited by two features: the support for class-based programming and the distinction between methods and attributes. These extensions are focused on making possible in Prolog to construct the type of programs typical of languages such as C++ and Java. In contrast, Logtalk emphasis is on predicate encapsulation and reusing solutions to cope with software engineering problems.

One of the most striking observations is that most object-oriented extensions do not seem to be used in-house for development or for extending the supporting Prolog compilers. For example, SICStus has a large set of libraries that use the module system, but no SICStus Objects versions of them. In fact, SICStus Objects is just another library. The same thing happens with O'CIAO. LPA recently added a TCP/IP library to WinProlog, but the new predicates were added as new built-ins instead of using its own Prolog++ object-oriented extension. The same can be said about other extensions. If the Prolog developers and vendors behind these object-oriented extensions do not appear to use them, why should their users?

Methods and attributes versus predicates

Most of the Prolog object-oriented extensions described above make a distinction between methods and attributes. In some of them, this distinction is made for programming convenience, as the same functionality could be accomplished, although in a more cumbersome way, using Prolog built-in predicates such as `assertz/1` and `retract/1`. Most extensions take advantage of the distinction between methods and attributes to optimize the code that is generated when compiling objects. In Logtalk, objects contain predicates, which can be either static or dynamic. No distinction is made between a predicate that represents the equivalent of a method, and a predicate that represents an attribute. Logtalk aims to provide Prolog with code encapsulation and reuse features based on object-oriented concepts, not to bring to Prolog the programming style typical of object-oriented languages such as C++.

Class-based versus prototype-based extensions

A Prolog object-oriented extension provides a competing encapsulation solution to the use of module systems, either as found on some Prolog compilers or as specified in the ISO Prolog standard. The extensions described above can be classified as either class-based or prototype-based. With the exception of SICStus Objects, all other extensions are class-based. From the point-of-view of Prolog programming, classes are not the best choice as prototypes provide a much fitter replacement for Prolog modules. Nevertheless, the preference for class-based designs is to be expected as this is the most widespread type of system found on object-oriented languages. Logtalk supports both prototypes

and classes. As such, it provides an alternative to the use of modules and, at the same time, enables the use of class hierarchies whenever necessary.

1.2.3 Prolog module systems

Like objects, modules provide us with a way of defining namespaces, instead of being limited to the flat database model of Prolog. There are a number of Prolog compilers implementing module systems and an ISO Prolog standard for modules. SICStus Prolog was chosen as a representative of the module systems as found in most Prolog compilers. Its module system is broadly compatible with the ones found on other compilers such as YAP or Quintus Prolog, and similar to the module system of other widely used compilers such as SWI-Prolog. A general description of the basic concepts of Prolog module systems is outside the scope of this thesis. It is assumed that the reader is familiar with these module systems and with the ISO Prolog standard for modules. As such, the descriptions below focus on the code encapsulation and reuse features of modules.

SICStus Prolog module system

The SICStus Prolog module system is broadly compatible with the module systems found on other Prolog compilers. Modules predicates are encapsulated between an opening and a closing directive. The opening directive lists the predicates that are exported by the module. Exported predicates are the equivalent to public object predicates. These predicates can be called without using module qualification after importing the module into the module where we want to use them. Modules do not provide data hiding; any module predicate can be called using explicit module qualification.

There is an interesting difference between how module predicates and object predicates are called. While a module predicate can be called without module qualification after importing a module, object predicates are called by sending an explicit message to an object. A detailed discussion of the pros and cons of explicit qualification to call an encapsulated feature can be found in [53].

ISO Prolog Standard module system

The ISO standard for Prolog modules [3] was published five years after the publication of the ISO standard for the Prolog core language [2]. The module system specified in this standard has some serious shortcomings that hampers its adoption by the Prolog community. Two of these shortcomings are described next. Two other important issues regarding declaration of metapredicates and separation of interface from implementation are discussed in Chapter 3 and in Chapter 4, respectively.

Encapsulation and data hiding Similar to objects, modules provide namespaces, enabling the encapsulation of predicates. However, unlike objects, modules do not support data hiding, a feature commonly associated to encapsulation mechanisms. In the section 6.2.2 — Procedure Visibility, the ISO standard for Prolog modules states:

> "All procedures defined in a module are accessible from any module by use of explicit module qualification. It shall be an allowable extension to provide a mechanism that hides certain procedures defined in a module M so that they cannot be activated, inspected or modified except from within a body of the module M".

Thus, modules provide a form of encapsulation where data hiding is optional and implementation dependent. Nevertheless, it should be noted that this limitation is in line with current practice regarding existing module systems.

Compatibility issues A standard should be based on current practice. In alternative, a standard may specify new features that are compelling enough so that current implementations are upgraded to conform to it. The ISO Prolog standard for modules fails in both accounts. The module system it specifies is incompatible with existing and widely used module systems. In addition, its new features do not overcome in any significant way the limitations of current module systems. This may explain why most Prolog vendors opted, until now, for ignoring it. This contrasts with the ISO Prolog standard for the core language that is being adopted by most Prolog compilers.

Performance of module systems versus object-oriented extensions

Encapsulating Prolog code in a Logtalk object implies necessarily an overhead compared to the usual Prolog flat database model. Sending a message to an object implies checking whether the object exists, verify whether the message is part of the object interface, and finding a predicate definition (a method) to compute the answer. Depending on whether these checks are performed at compile time or at runtime, the overhead can be similar to calling a predicate encapsulated in a Prolog module. In other cases, the overhead will be bigger due to the use of features such as inheritance and dynamic binding.

1.3 Working with objects

This section describes the syntax for defining objects and object hierarchies, the Logtalk built-in predicates for object handling, and the available object directives. See Appendix B for the full specification of the directives and built-in predicates.

1.3.1 Defining a new object

We can define a new object in the same way we write Prolog code: by using a text editor. Object code (directives and predicates) is textually encapsulated between two Logtalk directives: `object/1-5` and `end_object/0`. The simplest object will be a self-contained prototype, not depending on any other Logtalk entity:

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

The first argument of the opening directive is the object identifier. Object identifiers can be atoms or compound terms[1]. Objects share a single namespace with protocols (presented in Chapter 4) and categories (presented in Chapter 5): we cannot have an object with the same identifier as an existing object, protocol, or category.

[1] Compound terms are used as identifiers for parametric objects; these will be described later in this chapter.

1.3.2 Defining object hierarchies

In object-oriented programming, objects are typically organized in hierarchies. Object hierarchies enable sharing of interface and implementation through inheritance. Logtalk supports both prototype hierarchies and class hierarchies.

Prototype hierarchies

Prototype hierarchies are constructed by defining *extension* relations between objects. To define an object as an extension of one or more objects we will write:

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

The sequence of the parent prototypes in the object-opening directive determines the lookup order for predicate inheritance. The lookup is performed using a depth-first strategy.

Class hierarchies

Class hierarchies are constructed by defining *instantiation* and *specialization* relations between objects. To define an object as an instance of one or more classes, we will write:

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

To define a class as a specialization of one or more classes (its superclass), we will write:

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

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

```
:- object(Class,
    instantiates(Metaclasses),
    specializes(Superclasses)).
    ...
:- end_object.
```

Compiling objects

A stand-alone object 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 an instantiation or a specialization relation. The best way to accomplish this is to define a set of objects that provides the basis of a reflective system, as will be exemplified later in this

chapter. If a reflective system solution is not necessary, but we still want to construct class hierarchies, then we can simply turn a class into an instance of itself or, in other words, turn a class into its own metaclass. For example:

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

An alternative solution would be to add a predefined root class to Logtalk, and then to use that class as the default root class when defining class-based hierarchies. However, it is a Logtalk design choice not to specify any predefined entities in order to keep the language simple and unbiased to particular solutions. We can always use an entity library if necessary.

1.3.3 Inheritance

In the context of logic programming, inheritance can be interpreted as a form of theory extension: an object virtually contains, in addition to its own predicates, all the predicates inherited from other objects. In order to determine the set of predicates that characterizes an object, inheritance rules are applied to the object relations with other entities. These rules specify how predicates are inherited, what search algorithm to use when there is more than one inheritance path, and how inheritance conflicts are solved.

In languages, such as Logtalk, which support the separation between interface and implementation, a distinction is made between interface inheritance and implementation inheritance. In Logtalk, interface inheritance refers to the inheritance of predicate declarations, while implementation inheritance refers to the inheritance of predicate definitions. Therefore, two sets of inheritance rules are necessary: one set for interface inheritance and another for implementation inheritance.

The inheritance rules are applied whenever a message is sent to an object[2]. Interface inheritance rules are applied to find a predicate declaration matching the message in order to check its validity. Implementation inheritance rules are applied to find a predicate definition to answer the message.

Inheritance rules differ for prototype-based hierarchies and class-based hierarchies, as will be explained below. Nevertheless, some rules are common to both hierarchy types:

1. Local predicate definitions always override conflicting inherited definitions.

2. Whenever there is more than one inheritance path (for example, when using multiple inheritance), a depth-first search algorithm is applied. The first declaration (definition) found overrides any possible declarations (definitions) on any remaining inheritance paths.

The use of a depth-first searching algorithm (on the implementation of multiple inheritance mechanisms) is common to other Prolog object-oriented extensions such as L&O or Prolog++. This contrasts with the sophisticated algorithms found in some programming languages such as CLOS [54]. Nevertheless, no solution has been found that might

[2]This should be interpreted in a loose sense. Inheritance rules can be applied at compile time (static binding) or at runtime (dynamic binding). These options will be discussed in Chapter 8.

be considered satisfactory for all the problems raised by the multiple inheritance mechanisms. A detailed discussion of these problems can be found in [55]. Logtalk multiple inheritance support will be discussed later in this section.

Along with extension, instantiation, and specialization relations with other objects, an object may also implement one or more protocols, and import one or more categories. Protocols are Logtalk entities containing predicate declarations that can be implemented by any object (protocols will be fully discussed in Chapter 4). Categories are Logtalk entities containing predicate declarations and predicate definitions that can be imported by any object (categories will be fully discussed in Chapter 5). Inheritance rules work as if all the object protocol and category relations have been flattened by moving the contents of the implemented protocols and importing categories into the object. This flattening process complies with the following rules:

3. Local predicate declarations override conflicting predicate declarations *inherited* from protocols and categories. Predicate declarations *inherited* from protocols override conflicting predicate declarations *inherited* from categories.

4. Local predicate definitions override conflicting predicate definitions *inherited* from categories.

Note that this flattening process results in implemented protocols and imported categories being always searched before extended, instantiated, or specialized objects.

Logtalk support for multiple-inheritance, multiple protocol implementation, and multiple category importation implies an ordering of alternative inheritance paths per relation type, expressed by the following rule:

5. The ordering of alternative inheritance paths per relation type is the same as the order of the related entities in the corresponding clause in the object-opening directive (for example, if a prototype extends two parents, the first declared parent will be searched before the second one).

These five inheritance rules are complemented by rules that are specific to each type of object hierarchy, as will be explained next.

Inheritance rules for prototype-based hierarchies

The inheritance rules for prototype-based hierarchies are simple and similar for both interface inheritance and implementation inheritance. When a prototype does not contain a local predicate declaration matching a message, the search is done in the prototype parents. Likewise, when a prototype does not contain a local predicate definition for answering a message, the search is done in the prototype parents.

Inheritance rules for class-based hierarchies

The inheritance rules for class-based hierarchies must take into account the distinction between instantiation and specialization relations, which implies different rules for interface inheritance and implementation inheritance. When an instance receives a message, the search for a matching predicate declaration starts in the instance classes, and then proceeds to the class superclasses[3]. The search for a predicate definition in order to

[3]Note that a local predicate declaration does not override inherited predicate declarations because the search always starts at the instance classes, making the local declaration meaningful only for the descendant instances of the instance itself.

answer the message starts at the instance itself and, if not found, proceeds to the instance classes and respective superclasses. Note that this inheritance rule differs from the rules of common object-oriented languages where methods are always defined at the class level.

Multiple inheritance

Inheritance mechanisms can hardly be discussed without referring to the long, and probably endless, debate on single inheritance versus multiple inheritance. Single inheritance mechanisms can be implemented efficiently, but they impose several limitations on code reuse, even if the multiple characteristics we intend to inherit are orthogonal. In contrast, multiple inheritance mechanisms are attractive in their apparent capability of modeling complex situations. However, the use of multiple inheritance raises some complex problems in the domain of software engineering, particularly in code reuse and application maintenance. Multiple inheritance is often more useful as an analysis and project abstraction, than as an implementation technique [56]. All these problems are substantially reduced if we preferably use composition mechanisms instead of specialization mechanisms in software development [22]. As will be fully discussed in Chapter 5, Logtalk supports both the classical instance-based composition and a new composition mechanism based on the concept of *categories*. Categories support implementation reuse in the same way as protocols support interface reuse, providing an alternative solution to problems that would require a multiple inheritance solution on other programming languages. Nevertheless, Logtalk supports multi-inheritance for those rare occasions where its composition mechanisms fail to provide a suitable solution. This support, however, does not provide any mechanism for selecting a specific inheritance path or for dealing with inheritance conflicts.

1.3.4 Creating a new object at runtime

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

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

The first argument, the identifier of the new object (a Prolog atom or compound term), must not match any existing entity identifier. The second argument corresponds to the relations described in the opening object directive. The third and fourth arguments are lists of directives and predicate clauses, respectively. For example, the following call:

```
| ?- create_object(o1, [extends(o2)], [public(p/1)], [p(1), p(2)]).
```

is equivalent to compiling and loading the object:

```
:- object(o1,
    extends(o2)).

    :- dynamic.

    :- public(p/1).

    p(1).
```

```
    p(2).

:- end_object.
```

If we need to create many (dynamic) objects at runtime, then it is better to define a metaclass or a prototype with a predicate that calls this built-in predicate in order to create new objects. This predicate may provide automatic generation of object identifiers and accept object initialization options. The current Logtalk implementation contains example classes defining such predicates in its library.

1.3.5 Abolishing dynamic objects

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

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

The argument must be an identifier of an existent dynamic object; otherwise an error will be thrown.

1.3.6 Object directives

Object directives are used to set initialization goals, define object properties, and for documenting objects.

Object initialization

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

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

The initialization goal can be any valid Prolog or Logtalk call. For example, the goal can be a call to a locally defined predicate:

```
:- object(foo).

    :- initialization(init).

    :- private(init/0).

    init :-
        ... .
    ...

:- end_object.
```

or a message to other object:

```
:- object(assembler).

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

:- end_object.
```

The `::/2` operator is used in Logtalk for message sending. The initialization goal can also be a message to *self* in order to call an inherited predicate. Assuming, for example, that we have an object named `profiler` defining a `reset/0` predicate, we could write:

```
:- object(stopwatch,
    extends(profiler)).

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

:- end_object.
```

The `::/1` operator is used in Logtalk for sending a message to *self*. Note that, in this context, *self* denotes the object containing the directive.

Descendant objects do not inherit initialization directives from ancestor objects. In addition, note that by initialization we do not necessarily mean setting an object's dynamic state.

Dynamic objects

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.
```

As in most Prolog systems, dynamic code provides lower performance than static code. Therefore, we should only use dynamic objects whenever they need to be abolished during program execution.

Object dependencies

In addition to the relations declared in the object-opening directive, the predicate definitions contained in the object may imply other dependencies. These can be documented by using the directives `calls/1` and `uses/1`.

The `calls/1` directive can be used when a predicate definition sends some message that is declared in a specific protocol:

```
:- calls(Protocol).
```

Protocols will be discussed in Chapter 4.

When a predicate definition sends a message to a specific object, this dependence can be declared with the directive `uses/1`:

```
:- uses(Object).
```

These two directives may be used by the Logtalk runtime engine to ensure that all necessary entities are loaded when running an application. The directive `uses/1` is also the basis for a planned extension of the Logtalk language to support object namespaces. Both directives will be further discussed in Chapter 7.

Object documentation

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

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

Assuming, for example, that we have defined an object containing list predicates, it could be documented as follows:

```
:- info([
    version is 1.0,
    author is 'Paulo Moura',
    date is 2000/7/24,
    comment is 'List predicates.']).
```

This directive will be fully discussed in Chapter 7.

1.4 The pseudo-object `user`

The only predefined object in Logtalk is the pseudo-object `user`, which contains all user predicate definitions not encapsulated in a Logtalk entity. These predicates are assumed to be implicitly declared public. This pseudo-object is sometimes used in order to call built-in Prolog predicates that are redefined inside Logtalk objects.

1.5 Finding about objects

Logtalk provides a set of built-in predicates that enables reflective computations about objects and object relations in our applications.

1.5.1 Finding defined objects

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

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

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

1.5.2 Object relations

Logtalk provides a set of built-in predicates for querying the system about the possible relations an object may have with other objects.

Extension relations

The built-in predicate `extends_object/2` can be used to query all prototype extension relations:

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

Instantiation relations

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

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

Note that `Instance` can be a class with the respective `Class` being (one of) its meta-class(es).

Specialization relations

The built-in predicate `specializes_class/2` can be used to query all class specialization relations:

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

1.5.3 Object properties

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

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

An object may have either the `static`, `dynamic`, or `built_in` property. Dynamic objects can be abolished at runtime by calling the `abolish_object/1` built-in predicate.

1.6 Examples

This section presents three simple examples of Logtalk programs. These examples include both prototype-based solutions and class-based solutions. The full source code of these examples is available with the current distribution of Logtalk.

1.6.1 Towers of Hanoi

This example shows how to use an object to encapsulate a solution for the well-known "Towers of Hanoi" problem. The object will be a self-contained prototype with its interface resuming to a single predicate whose argument will be the number of disks for which we want to solve the problem:

```
:- object(hanoi).

    :- public(run/1).

    run(Disks) :-
        move(Disks, left, middle, right).

    move(1, Left, _, Right):-
        !,
        report(Left, Right).
```

```
    move(Disks, Left, Aux, Right):-
        Disks2 is Disks - 1,
        move(Disks2, Left, Right, Aux),
        report(Left, Right),
        move(Disks2, Aux, Left, Right).

    report(Pole1, Pole2):-
        write('Move a disk from '),
        writeq(Pole1), write(' to '), writeq(Pole2),
        write('.'), nl.

:- end_object.
```

Note that, if we remove the Logtalk directives (the opening and closing object directives, and the predicate directive), the remaining code consists of Prolog-compliant predicate clauses. This is an important feature of Logtalk: Prolog code can be easily encapsulated by Logtalk objects with little or no modifications.

The predicate directive `public/1` is used to declare predicates that can be called from outside the object through message sending. This predicate directive will be further discussed in Chapter 3.

After compiling and loading this object, we can test our code by sending the message `run/1` to the object. Message sending is performed using the infix operator `::/2`. An example call will be:

```
| ?- hanoi::run(3).

Move a disk from left to right.
Move a disk from left to middle.
Move a disk from right to middle.
Move a disk from left to right.
Move a disk from middle to left.
Move a disk from middle to right.
Move a disk from left to right.

yes
```

This example presents an object solution that is essentially equivalent to a module solution. In addition, it illustrates how, in Logtalk, we can easily define *stand-alone* objects that are not attached to any hierarchy, as in any prototype-based language.

1.6.2 A reflective class-based system

The easier and, at the same time, the most powerful way of working with instances and classes in Logtalk is to define a set of classes that will provide the basis for a reflective class-based system. This example extends the reflective class-based system presented in [57], by providing explicit support for abstract classes [27]. It is composed by three classes: `object`, `abstract_class`, and `class`. The class `object` works as the root of the inheritance graph, containing predicates common to all objects:

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

The class `abstract_class` works as the default metaclass for all abstract classes, containing predicates common to all classes:

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

The class `class` works as the root of the instantiation graph and the default metaclass for all instantiable classes, containing predicates for creating, initializing, and abolishing class instances:

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

Notice that all three classes are instances of the class `class`, including `class` itself, thus closing the potential infinite regression of metaclasses characteristic of reflective systems. The most remarkable feature of this set of classes is that each one inherits the predicates of itself and of the other two classes with no lookup loops. For instance, if we send a message to `object`, the search for a matching predicate declaration starts at its metaclass, `class`, continues at the metaclass superclass, `abstract_class`, and finally at `object` itself.

1.6.3 Geometric shapes

This is a classical object-oriented programming example. The idea is to represent geometrical shapes and their properties. In order to help compare the pros and cons of prototypes and classes, the geometric shapes will be implemented both as a prototype hierarchy and as a class hierarchy.

Prototype-based version

This first version represents geometric shapes through a hierarchy of prototypes. The root prototype, `shape`, declares common geometric shape properties, defining default values for them:

```
:- object(shape).

    :- public(color/1).
    :- public(position/2).

    color(blue).     % default shape color
```

```
    position(0, 0).  % default shape position

:- end_object.
```

We can define a prototype named `polygon`, containing properties common to all geometric polygons, as an extension of the prototype `shape`:

```
:- object(polygon,
    extends(shape)).

    :- public(nsides/1).
    :- public(side/1).

    :- public(area/1).
    :- public(perimeter/1).

    side(1).         % default side length

:- end_object.
```

Being a prototype, we can send to it any message corresponding to a predicate that it declares or inherits from its parent prototypes. For example:

```
| ?- polygon::area(A).

no
```

The query fails because the predicate `area/1` is not *defined*. Nevertheless, the query itself is valid because the predicate `area/1` is *declared* for the prototype `polygon` (and for all prototypes inheriting from it)[4].

Regular polygons can be defined as an extension of generic polygons. Given a regular polygon, we can define a default method for calculating its perimeter:

```
:- object(regular_polygon,
    extends(polygon)).

    perimeter(Perimeter) :-
        ::nsides(Number),
        ::side(Side),
        Perimeter is Number*Side.

:- end_object.
```

The operator `::/1` allows us to send a message to *self*, i.e. the object that receives the message `perimeter/1`. Therefore, in order to calculate the perimeter of a regular polygon, we send to *self* messages for retrieving its number of sides and its side length. So far, all prototypes represent abstract shape concepts. Actual shapes such as squares

[4]The semantics of message sending will be further discussed in Chapter 2.

and triangles can be represented as extensions of `regular_polygon`. For example:

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

    nsides(4).

    area(Area) :-
        ::side(Side),
        Area is Side*Side.

:- end_object.
```

We can ask the prototype `square` for any of the shape properties declared by itself and by its ancestors. For example:

```
| ?- square::nsides(N).

N = 4
yes
```

Some queries, such as `area/1` or `perimeter/1`, return results that are calculated using default property values (in this case, side length):

```
| ?- square::area(A).

A = 1
yes
```

Specific squares can be represented as extensions of the prototype `square`. For example, a prototype such as:

```
:- object(q1,
    extends(square)).

:- end_object.
```

will represent a square with the default values for polygon side length, shape color, and shape spatial position:

```
| ?- q1::(color(Color), side(Side), position(X, Y)).

Color = blue
Side = 1
X = 0
Y = 0
yes
```

The syntax employed on this query, `Object::(Msg1, Msg2, ..., Msgn)`, is a shorthand notation for sending a set of messages to the same object.

When defining a new prototype through extension of existing prototypes, any inherited predicate definition can be overridden with a local definition. For example, if we define a specific square with the following property values:

```
:- object(q2,
    extends(square)).

    position(2, 3).  % override default inherited values
    color(red).
    side(3).

:- end_object.
```

it will be the q2 specific value for side length that will be used when calculating its area and its perimeter:

```
| ?- q2::(side(Side), area(Area), perimeter(Perimeter)).

Side = 3
Area = 9
Perimeter = 12
yes
```

This simple example could be easily extended to represent other regular polygons and other types of geometric shapes, along with their properties. When using prototypes, there is no distinction between abstract shape concepts, such as "polygon", and actual shapes, such as "square". Nor is there a distinction between abstractions, such as "square", and concrete objects, such as"q1" or "q2". When these distinctions are important in our application, we should then opt for a class hierarchy instead. However, as shown in this example, prototypes allow us to query indistinctively abstractions and concrete objects. This is not possible with classes, as it will be explained next.

Class-based version

In this second version, geometric shapes are represented through a hierarchy of classes and instances. This version assumes that the classes presented in the reflective class-based system example have already been defined. In addition, since the contents of all the classes and instances are the same as the contents of the corresponding prototypes on the first version, the contents of the objects will be omitted unless they are necessary to clarify some concept. In fact, the differences between the prototypes and the corresponding classes and instances lie exclusively on the object-opening directives.

The root class, `shape`, is an abstract class and, as such, an instance of the class `abstract_class`. In addition, we may define `shape` as a specialization of the class `object`, which represents properties common to all objects, including geometric shapes. The resulting object definition will be:

```
:- object(shape,
    instantiates(abstract_class),
    specializes(object)).
    ...
:- end_object.
```

The class `shape` is specialized by the class `polygon`, also an abstract class:

```
:- object(polygon,
    instantiates(abstract_class),
    specializes(shape)).
    ...
:- end_object.
```

The class `regular_polygon`, a specialization of class `polygon`, is also an abstract class:

```
:- object(regular_polygon,
    instantiates(abstract_class),
    specializes(polygon)).
    ...
:- end_object.
```

The class `square` is an instantiable class, so its metaclass is the class `class`:

```
:- object(square,
    instantiates(class),
    specializes(regular_polygon)).
    ...
    nsides(4).
    ...
:- end_object.
```

Note that the predicate `nsides/1` works as a shared instance variable: it is a property whose value is shared by all `square` instances. Its definition is stored at class level in order to avoid repeating the same information in every instance. Shared instance variables are known in Java as static variables and in Smalltalk as class variables (a misleading name for a language that supports metaclasses).

In contrast with prototypes, we cannot send messages to classes, such as `square`, to retrieve shared shape properties. For example, the following query will throw an exception:

```
| ?- square::nsides(N).

! error(
    existence_error(predicate_declaration, nsides(_)),
    square::nsides(N),
    user)
```

This happens because the predicate `nsides/1` is declared for all descendant instances of the class `polygon`, not for the classes themselves. Defining a `square` instance such as:

```
:- object(q1,
    instantiates(square)).

:- end_object.
```

allows us to use any public predicate declared in the instance class, or in the instance class superclasses, as a message. For example:

```
| ?- q1::nsides(N).

N = 4
yes
```

Default property values stored in classes work in the same way as in the prototype version:

```
| ?- q1::(color(Color), side(Side), position(X, Y)).

Color = blue
Side = 1
X = 0
Y = 0
yes
```

We can also override default property values with local instance values, as we did with prototypes:

```
:- object(q2,
    instantiates(square)).
    ...
    side(3).
    ...
:- end_object.
```

The same queries will return the same answers, as expected:

```
| ?- q2::(side(Side), area(Area), perimeter(Perimeter)).

Side = 3
Area = 9
Perimeter = 12
yes
```

The distinction between abstract classes and instantiable classes, and between classes and instances, provides a rigid structure that reflects the distinction between abstract shapes and actual shapes, and between actual shapes and concrete objects. However, this rigid structure prevents us from querying classes using the same predicates that they declare and define for their instances. For example, we cannot ask `square` (or any other class representing an actual shape) about its number of sides. This query can only be sent to the class instances. But even so, we have no way of telling an answer that is specific to an instance from an answer that applies to all class instances.

1.7 Parametric objects

A parametric object is an object whose identifier is a compound term containing free variables. These variables play the role of object parameters. Object predicates can then be coded to depend on the parameter values. Thus, a parametric object can be regarded as a generic object from which specific *instantiations* can be derived by instantiating the

object parameters. Parameter instantiation usually takes place when sending a message to the object. Object parameters can be any valid Prolog term, including free variables, atomic terms, and compound terms.

Object parameters can be used to represent object state that is set when the object is created, but is not modified during the object lifetime, thus avoiding the use of destructive update methods for object initialization. Even when object state is updated at runtime, it may be possible to use the *instantiations* of the parametric object identifier to represent the history of object state changes. Moreover, parameter instantiation is undone on backtracking. Note that, when representing state by dynamic predicates, update operations using assert and retract predicates are not reversed on backtracking.

Parametric objects can also be used to attach a set of predicates to terms that share a common functor and arity. Thus, instead of using a term as a predicate argument, the predicate is called by sending the corresponding message to the term itself. This leads to a more data-driven programming style where data is represented by *instantiations* of parametric object identifiers.

1.7.1 Related work

Logtalk parametric objects are based on both L&O parametric theories and SICStus Objects parametric objects. Besides syntax differences, all three implementations are similar, with an important exception. As will be described later in this section, Logtalk parameter values are accessed via a built-in method. In L&O and SICStus Objects, parameters act as object global variables. There are two reasons for the Logtalk design choice. First, global variables are a foreign concept to the Prolog language. All variables in a predicate clause are local. Second, for complex objects, there is a risk of name conflict between parameter names and predicate local variables.

L&O parametric theories

In L&O, parametric objects are known as parametric theories. A theory is identified by a *label*, a term that is either a constant or a compound term with variables. One of the examples presented in [23] concerns description of trains:

```
train(S, Cl, Co):{
    colour(Cl).
    speed(S).
    country(Co).
    journey_time(Distance, T) :-
        T = Distance/S.
    }
```

The label variables are universally quantified over the theory. A specific train can be described by instantiating the label variables:

```
train(120, green, britain)
```

Messages can be sent to labels, which act as object identifiers. For example, the following message:

```
train(120, green, britain):journey_time(1000, Time)
```

will calculate a journey time using the value of the label first parameter as the speed of the train.

SICStus parametric objects

SICStus parametric objects are similar to L&O parametric theories, with parameters acting as global variables for the parametric object. The SICStus Objects manual [37] contains the following example, describing ellipses and circles:

```
ellipse(RX, RY, Color) :: {
    color(Color) &
    area(A) :-
        :(A is RX*RY*3.14159265)
    }.

circle(R, Color) :: {
    super(ellipse(R, R, Color))
    }.

red_circle(R) :: {
    super(circle(R, red))
    }.
```

SICStus Objects uses the predicate `super/1` to declare the ancestors of an object. This example illustrates parameter-passing between related objects in a hierarchy, a feature common to both L&O and Logtalk, which will be explained later in this section.

OL(P) object instances

OL(P) (Object Layer for Prolog) is a Prolog object-oriented extension that represents object instances using a notation similar to parametric objects. An instance `I` of an object named `Object` is represented as `Object(I)`. The term `I` is a list of attributes and attribute-value pairs. Instance state changes can be accomplished by constructing a new list with the updated and unchanged attributes. The OL(P) system documentation offers the following example:

```
| ?- rect(I)::area(A), rect(I)::move(5, 5, J).
```

The method `move/3` will return, in its third argument, the attribute list `J` resulting from the update of the attribute list `I`. In addition, OL(P) provides a nice notation for accessing and updating attributes. This solution for object state changes implies the use of extra arguments for methods that update attributes. Nevertheless, it is an important technique, which preserves the declarative semantics found on pure Prolog programs. We can easily apply this solution to Logtalk programs by using parametric objects. Moreover, we are not restricted to use a list of attributes. If the number of attributes is small, an identifier with the format `Object(V1, V2, ..., Vn)` will provide a more efficient solution.

C++ class templates

Although the notion of parametric objects may resemble the concept of C++ class templates, there is a fundamental difference: C++ class templates are not objects, we must *instantiate* them (by giving values to their parameters) to obtain a class and then instantiate the class to get a regular object. In addition, C++ templates must

be *instantiated* at compilation time while Logtalk object parameters are instantiated at runtime.

1.7.2 Accessing object parameters

In order to give access to object parameters, Logtalk provides the built-in local method `parameter/2`. It can be used as shown in the following template:

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

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

Arguments are numbered starting at one. For example:

```
:- object(ellipse(_RX, _RY, _Color)).

    ...
    color(Color) :-
        parameter(3, Color).
    ...
```

The built-in method `parameter/2` can be used to set, get, or test a parameter value. Assume, for example, that the following object has been compiled and loaded:

```
:- object(test(_Parameter)).

    :- public(gateway/1).
    :- public(transform/1).

    gateway(Parameter) :-
        parameter(1, Parameter).

    transform(Value) :-
        parameter(1, Parameter),
        double(Value, Parameter).

    double(Value, Double) :-
        ...

:- end_object.
```

For retrieving or setting the object parameter value we can write queries such as:

```
| ?- test(In)::gateway(abc), test(123)::gateway(Out).

In = abc,
Out = 123
yes
```

The predicate `gateway/1` is needed because the `parameter/2` is a *local* method and, as such, it cannot be used in message sending[5]. One may ask why use the predicate `gateway/1` in the first place when we could simply write something like:

```
| ?- test(Out).
```

The answer is that we may want to abstract the exact details of complex parameters whose structure might be changed later.

Object parameters can also be *computed* from the arguments of a message as in the following query:

```
| ?- test(Parameter):: transform(2).

Parameter = 4,
yes
```

Although in most cases the method `parameter/2` is used to retrieve parameter values, and not to set them, this example shows that *instantiations* of a parametric object can be generated by sending a message to the object. This built-in method will be further discussed in Chapter 3, along with alternative ways of accessing object parameters.

1.7.3 Parameter passing

The relationship between the parameters of an object and the parameters of its ancestor objects is established, in the object-opening directive, through unification. This ensures parameter passing between related objects at runtime. Parameter passing works with extension, instantiation, and specialization object relations.

As an example, consider the SICStus Objects example presented above, but using Logtalk syntax. A query such as:

```
| ?- red_circle(3)::area(Area).

Area = 28.274334
yes
```

will result in the following instantiation chain of the parametric object identifiers:

```
red_circle(3) -> circle(3, red) -> ellipse(3, 3, red)
```

Note that the predicate `area/1` is declared and defined in the object representing ellipses.

1.7.4 Examples

All the examples found on the L&O book and on the SICStus Objects manual can be easily translated to Logtalk syntax. Two more examples are presented below.

[5] Calls to the method `parameter/2` are compiled to a single unification call to ensure the best possible performance. This optimization would not be possible when calling the method via message sending.

Rectangles and squares

This first example is similar to the SICStus Objects example presented above. Instead of ellipses and circles, rectangles and squares will be used to illustrate parametric objects in Logtalk. Assume that we have a root prototype named `shape` defined as follows:

```
:- object(shape).

    :- public(nsides/1).
    ...

:- end_object.
```

Assume now that the relevant rectangle properties are its width and its height. Representing these two properties as object parameters, we can define an object named `rectangle(Width, Height)` as follows:

```
:- object(rectangle(_Width, _Height),
    extends(shape)).

    :- public(width/1).
    :- public(height/1).
    :- public(area/1).
    ...

    width(Width) :-
        parameter(1, Width).

    height(Height) :-
        parameter(2, Height).

    area(Area) :-
        ::width(Width),
        ::height(Height),
        Area is Width*Height.

    nsides(4).
    ...

:- end_object.
```

As illustrated by the object-opening directive, we can freely mix parametric and non-parametric objects in a hierarchy. Note that the object parameters, `_Width` and `_Height`, are anonymous variables. In order to use this object, we give values to its parameters. For example:

```
| ?- rectangle(4, 3)::area(Area).

Area = 12
yes
```

When a parametric object contains generic predicates that do not depend on parameter values, the corresponding messages can be sent to the object without first instantiating its parameters. For example:

```
| ?- rectangle(_, _)::nsides(N).

N = 4
yes
```

A square can be defined as a parametric object whose parameter will represent the length of its side, `square(Side)`. Moreover, a square is also a rectangle with equal values for width and height:

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

    :- public(side/1).

    side(Side) :-
        parameter(1, Side).

:- end_object.
```

With this object we can use all public predicates inherited from its parent prototype, `rectangle(Side, Side)`. For example:

```
| ?- square(2)::area(Area).

Area = 4
yes
```

Note that the predicates inherited from the object rectangle could be redefined in the object square for a better performance. In this case, only the predicate declarations would be inherited.

Symbolic differentiation of arithmetic expressions

This second example is based on a well-known example of symbolic differentiation and simplification of arithmetic expressions, which can be found on [58]. The L&O system also uses this example to illustrate parametric theories. The idea is to represent arithmetic expressions as parametric objects. We can regard an arithmetic expression as an object, whose name is the expression operator with greater precedence, and whose parameters are the operator sub-expressions (that are, themselves, objects). In order to simplify this example, the object methods will be restricted to symbolic differentiation of polynomials with a single variable and integer coefficients. In addition, we will omit any error-checking code. The symbolic simplification of arithmetic expressions could easily be programmed in a similar way.

For an arithmetic expression reduced to a single variable, x, we will have the following object:

```
:- object(x).

    :- public(diff/1). % returns the symbolic
                       % differentiation of self
    diff(1).

:- end_object.
```

Arithmetic addition, $x + y$, can be represented by the parametric object `'+'(X, Y)` or, using operator notation, `X + Y`. Taking into account that the operands can either be numbers or other arithmetic expressions, a possible definition will be:

```
:- object(_ + _).

    :- public(diff/1).

    diff(Diff) :-
        parameter(1, X), parameter(2, Y),
        diff(X, Y, Diff).

    diff(I, J, 0) :-
        integer(I), integer(J), !.

    diff(X, J, DX) :-
        integer(J), !, X::diff(DX).

    diff(I, Y, DY) :-
        integer(I), !, Y::diff(DY).

    diff(X, Y, DX + DY) :-
        X::diff(DX), Y::diff(DY).

:- end_object.
```

The object definitions for other simple arithmetic expressions, such as `X - Y` or `X * Y`, are similar. The expression x^n can be represented by the object `X ** N` as follows:

```
:- object(_ ** _).

    :- public(diff/1).

    diff(N * X ** N2) :-
        parameter(1, X), parameter(2, N),
        N2 is N - 1.

:- end_object.
```

After compiling and loading the above objects, and considering the operator precedences as specified in the ISO Prolog standard, any Prolog polynomial expression can be interpreted as an object. For example, the polynomial `2*x**3 + x**2 - 4*x` ($2x^3+x^2-4x$) will be interpreted as `(2*x**3) + (x**2 - 4*x)` ($(2x^3) + (x^2 - 4x)$). Thus, this expression is an *instance* of the `X + Y` parametric object. We can thus write queries such as:

```
| ?- (2*x**3 + x**2 - 4*x)::diff(D).

D = 2* (3*x**2)+2*x**1-4*1
yes
```

The resulting expression could then be symbolically simplified using a method defined in the same way as the differentiation method.

One problem with this example is that all objects contain a declaration for the predicate `diff/1`. This declaration could be moved to a *protocol*, a Logtalk entity that can contain predicate declarations that can be implemented by any object. This would avoid duplicating the declaration in each object that defines the predicate. Protocols will be introduced and discussed later, in Chapter 4.

1.8 Logtalk as a prototype language

This section summarizes Logtalk features as a prototype-based object-oriented language. It may be used for comparing Logtalk to other prototype programming languages. The classification method described in [59] is adopted here.

1.8.1 Object representation

In Logtalk, the main distinction between prototypes and classes is that prototypes are self-describing objects. The common view of prototypes as representing concrete (or prototypical) objects, as opposite to the view of classes as abstractions, is of secondary importance in Logtalk.

Logtalk prototypes are defined by a set of predicates, with no *a priori* distinction between methods and attributes. This is a feature shared by some prototype-based languages. For example, in Self [60], objects are composed by slots that can contain both data and procedures. In Logtalk, encapsulation is defined in a per-predicate basis by declaring the predicate scope to be either private, protected, or public.

1.8.2 Object creation and evolution

A prototype can be created either by extension of existing prototypes or from scratch (creation *ex-nihilo*). In both cases, the creation of a new prototype can be performed by compiling a source file containing the prototype textual representation or, at runtime, by using the Logtalk built-in predicate `create_object/4`, as explained in this chapter.

There is currently no built-in support for object creation by cloning an existing object, although Logtalk provides the necessary built-in predicates and built-in methods (described in Chapter 3) for defining a cloning method.

As far as object evolution is concerned, Logtalk supports the declaration, definition, and abolishing of object predicates at runtime, using object database built-in methods that will be described in Chapter 3.

1.8.3 Inheritance and life-time sharing between objects

Logtalk supports the definition of prototype hierarchies using an extension relation between prototypes. The extension relation (that is, the relation between a prototype and its parent prototypes) is hard-coded during prototype creation for performance reasons, preventing a straightforward implementation of dynamic inheritance (usually performed by changing parent links at runtime).

Logtalk supports both *life-time sharing* and *creation-time sharing* between a prototype and its parents, in a per-predicate basis. Therefore, modifications to a parent prototype may or may not be inherited by its descendant prototypes, depending on how the modification is performed.

1.8.4 Extensions, delegation and sharing

As will be discussed in Chapter 2, Logtalk does not provide a message delegation mechanism. Nevertheless, note that sending a message to *self* from within a prototype is equivalent to delegating the message to the parent prototypes.

In Logtalk, *property sharing* and *value sharing* are not a characteristic of the extension link between a prototype and its parents, but a consequence of how we use the Logtalk built-in methods for predicate modification to implement an assignment operation. In fact, there is no assignment primitive in Logtalk. What the language provides is a set of built-in methods that enable a predicate definition to be either abolished, retracted, or asserted. As methods, the object that gets modified is determined by the execution context of the assert and retract messages. Thus, we can use, in the same prototype, property sharing for some predicates and value sharing for other predicates. An example of both types of sharing is included in Chapter 3. This support for both property sharing and value sharing enables us to represent entities of the application domain by one or more prototypes. Thus, parent (extension) links may behave similarly to "is–a" links for some predicates but not for others.

1.9 Logtalk as a class-based language

This section summarizes Logtalk features as a class-based object-oriented language. It may be used for comparing Logtalk to other class-based programming languages.

1.9.1 Definition of classes and instances

Both classes and instances can be defined in a source file or created dynamically at runtime. Classes and instances created at runtime are always dynamic entities. Classes and instances defined in source files can be either dynamic or static. Dynamic entities can be abolished at runtime. Thus, working with instances does not necessarily imply runtime object creation. This contrasts to what happens with most object-oriented programming languages. Defining instances in a file may also be considered an unusual way of working with objects, but it complies with the Logtalk primary view of objects as encapsulation units.

1.9.2 Methods and variables

Logtalk object predicates subsume the concepts of object methods and variables. A detailed discussion on how to use predicates to easily implement concepts such as instance variables, shared instance variables, class, variables, instance methods, and class methods is carried out in Chapter 3.

1.9.3 Class interfaces

Logtalk supports the separation between implementation and interface through the definition of protocols. Protocols contain declarations of predicates that can be implemented by any object. Like objects, protocols are first-class entities in Logtalk. A class may implement several protocols and a protcol may be implemented by several classes. Protocols are presented in Chapter 4.

1.9.4 Component-based programming

Logtalk supports component-based programming through the definition of categories. Categories contain declarations and definitions of predicates that can be imported by any object. Like objects and protocols, categories are first-class entities in Logtalk. A class may import several categories and a category may be imported by several classes. Categories are presented in Chapter 5.

1.9.5 Class hierarchies

We can define single hierarchies (as in Smalltalk or Java) or multiple, independent class hierarchies (as in C++). These hierarchies can use either single or multiple-inheritance. In addition, an object may instantiate more than one class.

1.9.6 Metaclasses

In Logtalk, a class can be either an object (as in Smalltalk) or just a way of defining instances (as in C++). Logtalk supports both views of a class, at the same time, in the same application. Metaclasses can be defined for some classes or for all classes. We can thus define reflective systems as found in languages such as Smalltalk. In addition, metaclasses may be shared among classes or we may have one metaclass per class, similar to the Smalltalk "shadow" metaclass hierarchy.

1.9.7 Abstract classes

An abstract class is a class that cannot be instantiated. This may happen because some method is declared, but has its implementation deferred to some suitable subclass, or simply because the class describes some abstraction that does not directly represents concrete objects.

Most class-based languages, such as C++ and Java, support abstract classes. In C++, a class containing a pure virtual method cannot be instantiated. In Java, a class can be explicitly declared as abstract by the programmer using the keyword `abstract` (even if it does not contain any undefined method). Other languages such as Smalltalk have no mechanism for defining abstract classes. Logtalk uses an operational, rather than declarative, definition of abstract class: a class is abstract if does not recognize

any instance-creation message. Methods for instantiating classes are usually defined by the programmer, and are based on the Logtalk built-in predicate for creating objects described in this chapter (`create_object/4`). This built-in predicate knows nothing about abstract classes, being a basic building block for more feature-complete methods. As such, it is always possible to use this predicate for instantiating any class in Logtalk.

1.10 Summary

Logtalk views objects as a solution for the encapsulation and reuse of predicates. As such, the issues of dynamic state change, in the context of a logic programming language, are outside the scope of this thesis. Logtalk objects provide a way of organizing and reusing Prolog code, helping to solve the software engineering problems typical of large programs.

Logtalk objects can be defined in a source file or created dynamically at runtime. An object can be defined as either a static or a dynamic entity. For example, an instance can be defined as a static entity in a source file, in accordance with the primary view of objects as encapsulation units.

Logtalk view of objects as encapsulation units is carried further into the concepts of classes and prototypes. In Logtalk, the main difference between a prototype and a class is that a prototype is a self-describing object, while a class encapsulates predicates to be reused by other objects. That is, classes and prototypes imply different forms of predicate encapsulation and reuse.

As a Prolog object-oriented extension, Logtalk differentiates itself from other extensions first and foremost because of its broad compatibility with Prolog compilers and the ISO Prolog standard. Moreover, unlike some other extensions, Logtalk is not focused in bringing to Prolog the programming style typical of object-oriented languages such as C++ that evolved from imperative languages. As such, Logtalk does not provide destructive assignment primitives and does not make a distinction between attributes and methods. In addition, Logtalk is one of the few Prolog object-oriented extensions that is not a proprietary, commercial product.

As an object-oriented programming language, Logtalk supports several types of object systems. Logtalk supports at the same time, in the same application, the definition of classes and prototypes, of single and multiple independent hierarchies of objects (using single or multiple inheritance). Moreover, class hierarchies may define metaclasses for some of for all classes, enabling the construction of reflective systems. All types of object systems are supported in an equal basis, by the same set of language primitives. Logtalk does not favor one system type over another. As such, Logtalk can be viewed is a *neutral*, unbiased language. In addition, Logtalk language primitives (control constructs, built-in predicates, and built-in methods) may be used to build higher-level behavior for implementing other types of class-based systems.

Logtalk integration of classes and prototypes in a single language allows us to use the same message sending mechanisms and the same methods for dynamically creating, disposing, and enumerating objects. Although we cannot mix prototypes with classes and instances in the same hierarchy, we can freely exchange messages between them.

Logtalk prototypes can be used as a replacement for modules in Prolog programs, providing several important features not available in current Prolog module systems. Namely, prototypes provide data hiding, a feature missing in module systems. In ad-

dition, Logtalk provides a level of compatibility with Prolog compilers not matched by any module system. This makes a Logtalk program much more portable than a Prolog program that uses modules. Other important advantages of Logtalk objects over Prolog modules will be described in later chapters.

Chapter 2

Control constructs

Logtalk adds four new control constructs to those defined on the ISO Prolog standard: three control constructs for message sending and one control construct for calling external code, thus bypassing the compiler. All the control structures defined on the ISO standard can also be used in Logtalk as well.

This chapter begins by presenting the Logtalk message sending operators. Secondly, the control construct for calling external code is presented. Thirdly, the use of metapredicates as messages is discussed. Next, the message processing mechanism is explained. Finally, the Logtalk stance on, and alternative to the use of, message delegation is presented.

2.1 Message sending

Calling a predicate declared in an object interface is accomplished via message sending. Note that message sending is only the same as calling an object predicate if the object does not inherit predicate definitions from other objects. Otherwise, the predicate definition that will be executed may depend on the relationships of the object with other Logtalk entities.

2.1.1 Message sending operators

Logtalk uses the following three operators for message sending:

```
:- op(600, xfx, ::).    % message to an object
:- op(600,  fx, ::).    % message to self
:- op(600,  fx, ^^).    % super call
```

These operator definitions are compatible with the predefined operators described in the ISO Prolog standards. Some Prolog object-oriented extensions such as OL(P) [52] and SICStus Objects [24] also use the `::/2` operator for message sending, while others, such as L&O [23], use the `:/2` operator. However, this second operator is already defined in the ISO standard for the Prolog module system. Logtalk, like other systems, uses a different operator in order to prevent conflicts when using modules and objects in the same application.

2.1.2 Messages to objects

Sending a message to an object is accomplished by using the `::/2` infix operator:

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

The message must match a public predicate declared for the receiving object, a Logtalk built-in predicate, a Prolog built-in predicate, or a Prolog control construct, otherwise an error will be thrown (the appendix B contains a detailed description of the possible error messages).

The pattern `Object::Message` acts as a Prolog goal and it can be used as such with any ISO Prolog defined control constructs. For example, using the if-then-else Prolog control construct, we can write calls such as:

```
| ?- Obj1::Msg1 -> Obj2::Msg2; Obj3::Msg3.
```

Logtalk also adopts and extends the Prolog control constructs for conjunction and disjunction, providing some convenient syntactic sugar to send a set of messages to a set of objects, as described next.

2.1.3 Broadcasting

In Logtalk, broadcasting is interpreted as the sending of a message to a set of objects, sending of a set of messages to an object, or sending of a set of messages to a set of objects, adopting the definitions found in [23]. All these needs can be fulfilled by using the message sending method described in the previous section. However, for convenience, Logtalk implements an extended syntax for message sending that makes programming easier in those situations. This extended syntax uses the usual conjunction (`','/2`) and disjunction (`';'/2`) operators of Prolog, retaining their meaning. Note that this extended syntax exists for programmers convenience and does not necessarily imply any performance gains over sending one message to one object at a time.

Sending a set of messages to an object

To send a set of messages (as a conjunction of goals) to the same object, we can write:

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

This is semantically equivalent to:

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

When using disjunction we can write:

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

This is semantically equivalent to:

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

Sending a message to a set of objects

To send the same message to a set of objects we can write:

```
| ?- (Object1, Object2, ...)::Message.
```

This has the same semantics as:

```
| ?- Object1::Message, Object2::Message, ... .
```

We can also send the same message to a disjunction of objects by writing:

```
| ?- (Object1; Object2, ...)::Message.
```

This is semantically equivalent to:

```
| ?- Object1::Message; Object2::Message; ... .
```

Sending a set of messages to a set of objects

To send a set of messages to a set of objects, we have:

```
| ?- (Object1, Object2, ...)::Messages.
```

This is equivalent to:

```
| ?- Object1::Messages, Object2::Messages, ...
```

Likewise, typing:

```
| ?- (Object1; Object2; ...)::Messages.
```

is equivalent to:

```
| ?- Object1::Messages; Object2:: Messages; ...
```

Thus, in this case, broadcasting a set of messages to a set of objects reduces to the case of sending a set of messages to an object.

2.1.4 Messages to *self*

In defining a predicate, we often need to send a message to *self*, that is, to the same object that has received the original message. This is accomplished in Logtalk through the `::/1` prefix operator:

```
Predicate :-
    ...,
    ::Message,
    ...
```

This call can only be used in the body of a predicate definition. The broadcasting constructs described in the previous section can also be used combined with this operator as follows:

```
::(Message1, Message2, ...)
```

and, for disjunction:

```
::(Message1; Message2; ...)
```

A message must match a public or protected predicate declared for the receiving object (*self*), a private predicate declared in the object sending the message (*sender*), a Logtalk/Prolog built-in predicate, or a Prolog control construct, otherwise an error will be thrown (see the appendix B for details).

2.1.5 Calling redefined predicates

Sometimes, redefining an inherited predicate implies calling the inherited definition in the new code. For this purpose, Logtalk provides a control structure similar to the *super* primitive of Smalltalk, the `^^/1` prefix operator:

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

This call can only be used in the body of a predicate definition. Usually, its argument will be the predicate we are defining, but this is not strictly necessary.

Calling a redefined predicate in a new definition amounts to specialize an inherited definition, a concept similar to an object specialization by a descendant object. It is a key technique when reusing and adapting existing code.

Multiple inheritance and *super* calls

With single inheritance, there is, at the most, one inherited predicate definition to call using the *super* control construct. However, in case of multiple inheritance, there can be more than one inherited definition. The one that will be called by the *super* control construct is determined following the rules described in the previous chapter. In some object-oriented languages such as C++ [8] it is possible to specify in this context, whose inherited definition will be called by explicitly stating the class containing it. This is not possible in Logtalk.

2.2 Calling external code

Sometimes we need to call external Prolog code within an object. By external code, we mean code that is not encapsulated in other objects such as Prolog built-in predicates or user-defined code, other than Logtalk. For example, assume that we want to construct an object defining common comparison operations over fractions such as `</2`. If we write the corresponding predicate as:

```
N1/D1 < N2/D2 :-
    N1*D2 < N2*D1.
```

Logtalk will interpret this clause as a recursive definition as expected. However, what we want is to call the standard `</2` Prolog built-in predicate in the body of the predicate definition. For this code to work, we need a way to bypass the Logtalk compiler while compiling calls in a clause body. The ISO Prolog standard defines that a compound term with the functor `{}/1` can be expressed by enclosing its argument between curly brackets[1]. In Logtalk, we take advantage of this definition by using it as a control construct to bypass the compiler: all calls encapsulated between "{" and "}" are copied, unchanged, to the intermediate Prolog code that is generated when a Logtalk source file is compiled. We can then solve our problem easily by writing:

[1] The definition is probably intended as a hook for the representation of DCG clauses, although the standard does not mention them.

```
N1/D1 < N2/D2 :-
    {N1*D2 < N2*D1}.
```

An alternative solution will be to write:

```
N1/D1 < N2/D2 :-
    user::(N1*D2 < N2*D1).
```

However, this is less efficient (uses message sending instead of a direct call) and more verbose, hiding the fact that we are calling code that is not encapsulated on some object (`user` is just a pseudo-object containing all Prolog code not contained in some Logtalk entity).

Besides calling external code, the `{}/1` control construct can also be used to set the sender of a message to be other than the object from where the message is sent. For example, if we write:

```
foo(X) :-
    {obj::bar(X)}.
```

the sender of the `bar/1` message to the object `obj` will be the pseudo-object `user` instead of being the object encapsulating the predicate clause. This is an important technique in the context of event-driven programming that will be further discussed in chapter 6.

2.3 Control constructs and metapredicates as messages

As stated before, we can use any ISO Prolog standard defined control construct in Logtalk code as a message or as a call in the body of an object predicate clause. For example the goal:

```
| ?- Object::call(Goal).
```

is equivalent to writing:

```
| ?- call(Object::Goal).
```

We can also use any Logtalk or Prolog built-in predicate as a message. Using a built-in predicate other than a metapredicate results in a direct call of the predicate if it is has not been redefined for the object receiving the message. The case of Logtalk or Prolog built-in metapredicates, that are also built-in object methods, is similar to using control constructs as messages[2]. For example, if we use the `findall/3` built-in Prolog meta-predicate as a message:

```
| ?- Object::findall(Variable, Goal, List).
```

the call can also be written as:

```
| ?- findall(Variable, Object::Goal, List).
```

Logtalk compiles control constructs and built-in metapredicates used as messages in the body of object predicate clauses by moving the message sending inside the control construct or metapredicate, as exemplified above.

[2]Note that the distinction between a control construct and a meta-predicate, found in the ISO Prolog standard, is somewhat artificial: any control construct can be seen as a metapredicate.

2.4 Message processing

Processing a message sent to an object (including *self*) involves four steps:

1. Finding the predicate declaration corresponding to the message.
2. Checking the declaration for message scope validity.
3. Finding the predicate definition (or method) to answer the message.
4. Executing the predicate definition.

In Logtalk, the lookup of the predicate declaration and predicate definition is performed at runtime. In object-oriented programming, this is known as *dynamic binding*. When the lookup is performed at compile time, we talk about *static binding*. There are pros and cons to this design decision. On the pros side, we gain a very flexible development workflow: a change in one object does not imply recompilation of all derivated and related objects. This follows the typical Prolog development cycle where we often consult a file, run and debug our predicates, and then reconsult the fixed code. On the cons side, there is a performance penalty when compared to languages that support static binding. When using single inheritance, the lookup cost is, in the worst case, linear to the height of the inheritance tree. Binding messages to methods at compile time allows us to optimize message sending by eliminating all but the last step in message processing. It also allows us to catch some errors such as invalid messages. The Logtalk language specification does not prevent the implementation of a compiler that uses, whenever possible, static binding instead of dynamic binding. The drawbacks are a less flexible development environment (changes in one file may imply recompilation and reloading of several files) and maybe some restrictions on heavy self-modifying programs (redefining an object may invalidate some optimizations in dependent objects).

2.4.1 Execution context

The execution context of an object predicate (method) consists of the following data:

- The *sender* of the message (always an object).
- The object containing the predicate definition under execution, *this*.
- The object that received the original message under processing, *self*.

All the execution context information can be explicitly accessed (and used in the definition of predicates) by the programmer by calling a set of built-in local methods that will be described in the next chapter. The *sender* information is used in the implementation of metapredicates and meta-calls. When using parametric objects, the value of each parameter can be accessed via *self*.

The definition of *self* is the same as the one found in other Prolog object-oriented extensions and in common object-oriented languages such as Smalltalk. Logtalk shares the definition of *this* with OL(P), but this definition is different from the meaning of the keyword in C++, where it represents the same concept as *self*.

2.4.2 Closed-world assumption

The ISO Prolog standard specifies (in its section 7.5) that calling a declared predicate with no defined clauses should simply fail without raising an error[3]. This is sometimes called the *closed-world assumption*: what we do not specify as true is considered false. We adopt the same assumption in Logtalk: sending a valid message to an object for which there is no defined method fails without any error. In the ISO standard, a predicate is declared, if there is one dynamic, discontiguous, or multifile directive, which specifies it. In Logtalk, an object predicate is declared if there is a scope directive fir it, as will be discussed in the next chapter.

2.4.3 Exception handling

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. When those conditions are not met, an exception will be thrown. For example:

```
| ?- unknown::any.

! error(
    existence_error(object, unknown),
    unknown::any,
    user)
```

Exceptions while processing a message may also result from the execution of a predicate definition in response to the message. For example:

```
| ?- foo::abolish(bar/1).

! error(
    permission_error(modify, private_predicate, bar(_)),
    foo::abolish(bar/1),
    user)
```

In all cases, the exception terms thrown while processing a message have the following format:

```
error(Error, Message, Sender)
```

A complete and detailed description of the Logtalk control constructs exceptions can be found in Appendix B.

2.5 Message delegation

Message delegation is a form of message sending where the method selected to answer a message is executed in the context of the sender object, instead of being executed in the context of the receiver object. In the context of a prototype hierarchy, message delegation is equivalent to sending a message to *self*, whenever the receiver object is

[3]Note that this is different scenario from calling an unknown predicate: a declared predicate is known by the system.

a parent of sender object. Message delegation can be found on some prototype-based languages, including the SICStus Objects Prolog object-oriented extension. Logtalk does not support message delegation outside the scope of prototype hierarchies. Instead, the type of code reuse enabled by message delegation is accomplished in Logtalk through the use of *categories* (a category is a Logtalk entity that will be presented in Chapter 5). Nevertheless, Logtalk supports a limited form of message delegation through the definition of object metapredicates, as it will be discussed in Chapter 3.

2.6 Summary

A Logtalk object encapsulates predicate definitions. These predicate definitions can be called by sending a message to the object. Logtalk provides the usual message sending mechanisms found in other object-oriented languages, including sending a message to the object that received the original message under processing (*self*), and calling an inherited definition when redefining a predicate. Logtalk also provides a control construct that enables us to bypass the compiler when compiling a predicate definition.

Logtalk message sending mechanisms enforce the predicates declared scope, allowing only the use of visible predicates as messages, preventing the misuse of encapsulated private predicates. This basic feature, common to almost all object-oriented languages, is absent from the ISO standard for the Prolog module system, where we can call any encapsulated predicate as long as we know its name.

Chapter 3

Predicates

In Prolog, predicates are stored on a flat database and are used to describe what is true about the application domain. In Logtalk, predicates are encapsulated inside objects and are used to describe what is true about an application object (or set of objects). Logtalk predicates represent both object state and object behavior.

When describing Logtalk predicates, we need to make a distinction between a predicate declaration and a predicate definition. This distinction is necessary because Logtalk encapsulates predicates inside objects. Access to an object predicate is controlled by its declaration, a set of predicate directives, similar to Prolog predicate directives, which declare predicate properties such as scope, compilation mode (static or dynamic), and metapredicate arguments (if any). A predicate definition is simply a set of clauses for the predicate encapsulated inside an object or category. Both predicate declarations and definitions can be encapsulated inside objects and categories, while protocols can only contain predicate declarations. Protocols are presented in Chapter 4, while categories will be discussed in Chapter 5.

This chapter begins by describing how to declare and define object predicates. Secondly, it explains how to redefine and specialize object predicates in descendant objects. Next, the Logtalk built-in object predicates (built-in methods) and built-in predicates are described. Finally, several examples are presented, illustrating the use of object predicates to represent the notions of object state and object behavior common in other object-oriented languages.

3.1 Predicate declarations

All object (or category) predicates we want to access from other objects must be explicitly declared. A predicate declaration must contain, at least, a scope directive (described below). Other directives may be used for documenting a predicate or to ensure the proper compilation of predicate definitions. Predicates that are only used locally, inside an object, and are called exclusively from other object predicates, do not need to be declared.

3.1.1 Definitions

We start by defining a set of terms useful for the predicate classification that will be used throughout this thesis:

Local predicate A local predicate is a predicate that is defined inside an object (or a category) but that is not explicitly declared in a scope directive (described below).

Visible predicate A predicate that is declared for an object, a built-in method, a Prolog built-in predicate, or a Logtalk built-in predicate. The set of an object visible predicates is the set of predicates that can either be called from inside the object or used as messages to the object.

Metapredicate A predicate where one (or more) of its arguments will be called as a goal. For example, the predicate `findall/3` is a Prolog built-in metapredicate.

Private predicate A predicate that can only be called from inside the object where it is declared.

Protected predicate A predicate that can only be called from the object containing the predicate declaration or from a descendant object that inherits the predicate declaration.

Public predicate A predicate that can be called from any object.

3.1.2 Scope directives

As described above, a predicate can be public, protected, or private. A predicate is identified by its name and number of arguments, using the familiar Prolog predicate indicator notation `<name>/<nargs>`. The scope declarations are made using the directives `public/1`, `protected/1`, and `private/1`. For example:

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

:- protected(valid_init_option/1).

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

Several predicates can be declared in a single directive by using an extended syntax found in many Prolog compilers and formalized in the ISO standard: the directive argument can be a predicate indicator, a list of predicate indicators, or a predicate indicator sequence. For example:

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

:- protected([valid_init_option/1, valid_release_option/1]).

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

Note that we do not need to write scope directives for all defined predicates. Predicates not described by a scope directive (either local or inherited) are assumed local. Local predicates are invisible to the built-in reflection methods (described later on) and to the message and event handling mechanisms.

There are some fundamental differences between Logtalk object predicates and ISO Prolog module predicates [3] that should be stressed here. First, the meaning of public and private predicates follows the usual definitions of public and private methods found

in most object-oriented programming languages. Unfortunately, the ISO Prolog module system standard committee, choose to use these two terms to state whether the predicate definition can be accessed using the `clause/2` built-in predicate. Moreover, any module predicate can be called from any other module by the use of explicit module qualification. No predicate is private as long as we know its name. This is not possible in Logtalk, where the message sending mechanisms check and enforce the predicate scope directives.

3.1.3 Mode directive

Most predicates cannot be called with arbitrary arguments. In addition, most predicate arguments cannot have arbitrary instantiation modes. Predicate mode information is used in the Prolog ISO standards [2, 3] and in most Prolog compiler manuals to document built-in predicates. In Logtalk, the valid arguments and instantiation modes can be documented by using the directive `mode/2`. For example:

```
:- mode(member(?, +), zero_or_more).
```

The first argument describes a valid calling mode. The minimum information will be the instantiation mode of each argument. There are four possible values (described in [2]):

`+`
: Argument must be instantiated.

`-`
: Argument must be a free (non-instantiated) variable.

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

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

These four mode atoms are also declared as prefix operators by the Logtalk compiler. This makes it possible to include type information for each predicate argument. We can then rewrite the example above as:

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

Some of the possible type values are:

- `object`, `category`, `protocol`
- `event`
- `term`, `nonvar`, `var`
- `callable`, `compound`, `list`
- `atomic`, `atom`
- `number`, `integer`, `float`

The first four values are Logtalk specific. The remaining values are common Prolog types. We can also define our own types that can either be atoms or compound terms. For example, we can use the compound term `list(integer)` to denote an argument

that should be a list of integer values. It is thus possible to write in Logtalk mode declarations derived from those found in the ISO standard and in many Prolog compiler manuals.

The second directive argument documents the number of proofs (or solutions) for the specified mode. Meaning, the predicate deterministic type for the specified call mode. The possible values are:

`zero`
: Predicate always fails.

`one`
: Predicate always succeeds once.

`zero_or_one`
: Predicate either fails or succeeds once.

`zero_or_more`
: Predicate may fail or succeed one or more times.

`one_or_more`
: Predicate succeeds at least once.

`error`
: Predicate will throw an error.

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

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

A possible extension would be to use a compound term `error/1` for the second argument instead of the `error` atom, enabling the specification of the error term that will be thrown. For example:

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

However, this was deemed too cumbersome at the time the directive was designed. More importantly, not all error terms can be completely specified this way. For example, the first argument of the `arg/3` predicate must be an integer, otherwise the error term `type_error(integer, N)` should be thrown. Writing a mode declaration such as:

```
:- mode(arg(+, -, +), error(type_error(integer, N))).
```

is not an acceptable solution. In fact, it is not clear whether the variable `N` refers to the first argument of the predicate or not and, even if we include type information (there is only one integer argument), the use of a non-instantiated variable occurring only once does not make sense.

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

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

The current Logtalk implementation just parses and then discards this directive. However, it is possible (easy in fact) to modify the current implementation of the Logtalk

compiler to output mode directives suited for a Prolog compiler that would take advantage of them. Mode directives may also be used for type-checking method arguments in a future Logtalk release (an argument type information could be interpreted as the functor of a predicate used for testing actual argument values). Nevertheless, the use of mode directives is a good starting point for the documentation of object predicates.

A bit of history

A predicate instantiation mode directive, `mode/1`, was first introduced in the DEC-10 Prolog system [61, 62]. In this system, mode directives are used to improve performance, by allowing the compiler to produce more compact code (storage was a big issue at that time). Few of the Prolog compilers that are actively maintain today, support mode directives. One of them is ECLiPSe [63]. ECLiPSe mode directives are used to produce more compact and efficient code. Unlike Logtalk, in the DEC-10 and ECLiPSe compilers, mode directives only allow the specification of the predicate arguments' instantiation modes, but not the predicate deterministic type. The XSB logic programming system [64] also supports mode directives, but the compiler uses them only as tabling directives. The Mercury language [65, 66], also a logic programming language, supports mode directives similar to Logtalk's but uses a different, although equivalent set of atoms to declare the number of solutions: `erroneous` (`error`), `det` (`one`), `multi` (`one_or_more`), `failure` (`zero`), `semidet` (`zero_or_one`), and `nondet` (`zero_or_more`) [67]. Mercury mode directives are used by the compiler to improve code generation and optimization.

Most Prolog compilers such as YAP [68], Quintus Prolog [36], SICStus Prolog [24], SWI-Prolog [69], or BinProlog [70] accept mode directives, but only for documenting predicates and for compatibility with other Prolog compilers. The predicate call mode information is not used by the compiler to optimize code.

Some compilers define additional modes to provide further information about a predicate behavior. For example, in the Quintus Prolog mode declaration, the atom "`*`" signals a nondeterministic output argument while the atom "−" represents a deterministic output argument. Furthermore, Quintus uses "`+`" as the ISO "`@`" and defines two extra mode atoms for input arguments ("`+*`" and "`+−`"). The reason for of all these extra mode atoms is to provide useful information about the deterministic nature of an argument. Logtalk allows determinism to be declared in a per-call mode basis instead of at the argument level. It is interesting to note that the specification of all the built-in predicates in the ISO standard only requires four mode atoms.

3.1.4 Metapredicate directive

In Prolog, predicates that have arguments that will be called as goals, are named *metapredicates*. Those arguments are named *meta-arguments*. When we encapsulate a metapredicate inside a Logtalk object, the meta-arguments must be called in the context of the object that sends the message invoking the metapredicate. To ensure that these calls will be executed in the correct context, we need to use the `metapredicate/1` directive. For example, the Prolog built-in predicate `findall/3` would be declared as a metapredicate by writing:

```
:- metapredicate(findall(*, ::, *)).
```

The predicate arguments in this directive have the following meaning:

`::`
: Meta-argument that is called as a goal.

`*`
: Normal argument.

This is similar to the declaration of metapredicates in the ISO Prolog Standard for modules, except we use the atom "`::`" instead of "`:`", in order to be consistent with the message sending operators.

This directive must be included in every object containing a definition for the described predicate, even if the predicate declaration is inherited from another entity[1]. This will ensure the proper compilation of meta-arguments.

Example: tracing predicate calls

The following example is adapted from the ISO Prolog standard for the module system [3]. The idea is to define an object containing a predicate that can be used to trace the calls of predicates defined in other objects. By tracing, we mean printing the call term before and after execution. If we were to name the object and the predicate, respectively, `tracer` and `trace/1`, the corresponding Logtalk code would be as follows:

```
:- object(tracer).

    :- public(trace/1).
    :- metapredicate(trace(::)).

    trace(Goal) :-
        write('call: '), writeq(Goal), nl,
        call(Goal),
        write('exit: '), writeq(Goal), nl.

    trace(Goal) :-
        write('fail: '), writeq(Goal), nl,
        fail.

:- end_object.
```

Note that the `trace/1` predicate must be compiled in such way as its argument would be called in the context of the object sending the message to the `tracer` object.

In order to test the `tracer` object and the `trace/1` metapredicate, let me define an object that implements the quicksort algorithm. The sort code is adapted from an example in the SICStus Prolog User Manual [71]. It sends a `trace/1` message to the `tracer` object, enabling users to trace the calls to the `sort/2` and `partition/4` predicates, to learn how the quicksort algorithm works. Here is the Logtalk version:

```
:- object(sort(_Type)).

    :- public(sort/2).
```

[1] This is necessary because, in Logtalk, each entity is compiled independently from other entities.

```
    sort([], []).

    sort([Head| Tail], Sorted) :-
        tracer::(
            trace(partition(Tail, Head, Small, Large)),
            trace(sort(Small, Sorted1)),
            trace(sort(Large, Sorted2))),
        list::append(Sorted1, [Head| Sorted2], Sorted).

    partition([], _, [], []).

    partition([Head| Tail], Pivot, Small, Large) :-
        parameter(1, Type),
        (   Type::(Head < Pivot) ->
            Small = [Head| Small1], Large = Large1
        ;   Small = Small1, Large = [Head| Large1]
        ),
        partition(Tail, Pivot, Small1, Large1).

:- end_object.
```

The object parameter allows us to specify the type of the list elements by instantiating it to an object defining the usual comparison predicates. An example of a call and its output would be:

```
| ?- sort(user)::sort([3, 1, 4, 2, 9], Sorted).

call: partition([1,4,2,9],3,_358,_359)
exit: partition([1,4,2,9],3,[1,2],[4,9])
call: sort([1,2],_740)
call: partition([2],1,_967,_968)
exit: partition([2],1,[],[2])
call: sort([],_1300)
exit: sort([],[])
call: sort([2],_1539)
call: partition([],2,_1765,_1766)
exit: partition([],2,[],[])
call: sort([],_2093)
exit: sort([],[])
call: sort([],_2332)
exit: sort([],[])
exit: sort([2],[2])
exit: sort([1,2],[1,2])
call: sort([4,9],_2831)
call: partition([9],4,_3058,_3059)
exit: partition([9],4,[],[9])
call: sort([],_3391)
exit: sort([],[])
```

```
call: sort([9],_3630)
call: partition([],9,_3856,_3857)
exit: partition([],9,[],[])
call: sort([],_4184)
exit: sort([],[])
call: sort([],_4423)
exit: sort([],[])
exit: sort([9],[9])
exit: sort([4,9],[4,9])

Sorted = [1,2,3,4,9]
yes
```

In this example, the object parameter used is the pseudo-object `user`, which represents the Prolog database. This implies that the list item comparisons are performed using the standard Prolog built-in comparing predicates.

Metapredicate arguments as goal functors

Sometimes a metapredicate argument is not a goal but a functor or a compound term that will be used to construct a goal. The metapredicate `apply/2` is a familiar example. The first argument of this predicate is a compound term that is extended with a list of extra arguments, the predicate second argument. Assuming that we want to define a `meta` object encapsulating the usual predicate definition, we could write something like:

```
:- object(meta).

    :- public(apply/2).
    :- mode(apply(+callable, +list), zero_or_more).
    :- metapredicate(apply(::, *)).

    apply(Pred, Args) :-
        (atom(Pred) ->
            Goal =.. [Pred| Args]
            ;
            Pred =.. Old,
            list::append(Old, Args, New),
            Goal =.. New),
        call(Goal).

:- end_object.
```

However, this code does not work as intended because the argument of the metacall is not an argument of the metapredicate `apply/2`. The workaround is to define an auxiliary private metapredicate that will be called from `apply/2`. This auxiliary predicate contains an extra argument that will be bound to the goal constructed in the previous definition of `apply/2`:

```
:- object(meta).
```

```
    :- public(apply/2).
    :- mode(apply(+callable, +list), zero_or_more).

    :- private(apply/3).
    :- mode(apply(+callable, +list, -callable), zero_or_more).
    :- metapredicate(apply(*, *, ::)).

    apply(Pred, Args) :-
        apply(Pred, Args, _).

    apply(Pred, Args, Goal) :-
        (atom(Pred) ->
            Goal =.. [Pred| Args]
            ;
            Pred =.. Old,
            list::append(Old, Args, New),
            Goal =.. New),
        call(Goal).

:- end_object.
```

This code will work as expected because the *sender* context information will be passed from the call of `apply/2` to the call of `apply/3`. We can use the same workaround for other common metapredicates such as `call/N` or mapping predicates.

Dynamic metapredicates

It is not possible to declare, at runtime, new dynamic metapredicates. That is because we can declare a new (dynamic) predicate by asserting a clause for it but we cannot assert a metapredicate directive (or any other directive for that matter). However, we can always assert new clauses for dynamic metapredicates that are declared at compilation time.

Metapredicates in the ISO Prolog standard

One of the stated goals of the ISO standard is "to promote the applicability and portability of Prolog modules (...)" (section 1 — Scope). The portability goal seams to be forgotten when dealing with metapredicates. The standard proposal is to have a `colon_sets_calling_context` boolean flag to tell the programmer if access to the calling context when programming metapredicates can be done using the `:/2` operator. Worse, if the flag value is `false`, then the mechanism for accessing the calling context is left to the implementation! Therefore, the metapredicates that need to access the calling context will only be portable to Prolog compilers either where the flag have the value `true` or where the alternative mechanism of accessing the calling context is the same. The syntax for the declaration of meta-predicates also depends on the value of the `colon_sets_calling_context` flag (section 6.1.1.4 — Metapredicate mode indicators). If the flag is true, then the metapredicate mode indicator is a compound term declaring whose arguments are meta-arguments. However, when the flag value is

false, the metapredicate mode indicator resumes to a predicate indicator in the form `Functor/Arity`! Another roadblock to Prolog programs portability.

Metapredicate calls as message delegation

Calling a metapredicate through message sending can be interpreted as a limited form of message delegation. This interpretation follows from the fact that metacalls (in a metapredicate definition) are always executed in the context of the sender object. However, messages to *self* in the body of the metapredicate definition are executed with the *self* context information set to the receiver object. Contrary to what happens when delegating a message, the value of *self* in the execution context of a metapredicate is not set to the value of *self* of the execution context of the predicate responsible for sending the message (that triggers the call to the metapredicate).

3.1.5 Discontiguous directive

The ISO Prolog standard discontiguous directive allows predicate clauses to be discontiguous in a source file. In the same way, object predicate clauses may not be contiguous. In that case, we must declare the predicate discontiguous by using the `discontiguous/1` directive:

```
:- discontiguous(Functor/Arity).
```

With this directive, we can use the same extended syntax described for the scope directives. However, this is a directive that we should avoid to use because it makes our code harder to read.

This directive must be included in every object containing a discontiguous definition for the described predicate (even if the predicate declaration is inherited from other entity).

3.1.6 Dynamic directive

Similar to Prolog predicates, an object predicate can be either static or dynamic. By default, all object predicates are static. To declare a dynamic predicate we must use the `dynamic/1` directive:

```
:- dynamic(Functor/Arity).
```

Declaring dynamic predicates does not imply the use of dynamic objects. We can declare and define any number dynamic predicates inside a static object.

This directive must be included in every object containing a definition for the described predicate (even when the predicate declaration is inherited from other entity). If we do omit the dynamic declaration, the predicate definition will be compiled as static code, even if we are redefining an inherited dynamic predicate. Note that a static object is free to declare and define dynamic predicates.

When compared to Prolog dynamic predicates, the semantics of the dynamic object predicates are necessarily more elaborated, due to inheritance relations between objects. This semantics will be fully discussed later on this chapter, in the section on Logtalk built-in methods for object predicate database handling.

3.1.7 Documenting directive

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

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

The first argument identifies the predicate. The second argument is a list of `Key is Value` terms. For example, the predicate `run/1` declared in the "Towers of Hanoi" example on Chaper 1 could be documented as follows:

```
:- info(run/1, [
    comment is 'Prints the problem solution for n disks.',
    argnames is ['Disks']]).
```

This directive will be further discussed in Chapter 7.

3.1.8 Redeclaration of inherited predicates

When redeclaring an inherited predicate we are overriding it with the new declaration. As consequence, any inherited predicate definition will no longer be available for the descendant objects. The only sensible case where we may want to redeclare a predicate, is when we need to change its scope. That is, redeclaring a predicate allows us to change the scope of inherited predicates in a per predicate basis. To change the scope of all inherited predicates at the same time, we can use protected or private inheritance, as explained below.

3.2 Predicate definitions

As seen in most examples presented so far, we define object predicates as we have always defined Prolog predicates, with the only difference that we have four more control structures to play with (three message sending operators and an external call operator). For example, if we wish to define an object containing common list predicates such as `append/3` or `member/2` we could write:

```
:- object(list).

    :- public(append/2).
    :- 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, apart from the opening and closing object directives and the scope directives, what we have written above is plain Prolog code. Calls in a predicate clause body are compiled as calls to locally defined predicates, unless we use the message sending operators or the external call operator. This allows an easy conversion from Prolog code to Logtalk objects. For that, we only need to add the necessary encapsulation and scope directives to the existing code. In other object-oriented extensions to Prolog such as OL(P) [52] and SICStus Objects [24], calls in a predicate clause body must be prefixed with the message sending operator in order to call the local definition. For example, rewriting the object above for SICStus Objects results in the following code:

```
list :: {

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

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

}.
```

The consequence is a more verbose code and the need to edit existing Prolog code when encapsulating it inside objects.

3.3 Redefinition of inherited predicates

Inherited predicates can be redefined in a descendant object. The redefinition can be a restriction of the scope of the inherited predicates or a redefinition of the inherited predicate definitions. To restrict the scope of all inherited predicates we use public, protected, and private inheritance. To redefine an inherited predicate definition we add a local definition, which either overrides or specializes the inherited definition.

3.3.1 Public, protected, and private inheritance

Object extension, instantiation, and specialization relations are, by default, public. Thus, all predicates maintain their scopes when inherited. We can also define protected and private object relations in order to restrict the scope of inherited predicates. This is performed by prefixing, in the object-opening directive, the related object name with the corresponding scope keyword. The most common use for protected and private inheritance is to hide inherited protocol from object clients.

Logtalk implementation of public, protected, and private inheritance is similar to C++ [8] and Java [9]. This concept is extended in Logtalk in order to also be applied to protocol hierarchies, to protocol implementation by categories, and to object import relations with categories, as will be explained in chapters 4 and 5.

In order to illustrate this feature, the object extension relation will be used. The same syntax applies to the instantiation and specialization relations. To make all public predicates declared via an extended object become protected, we write:

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

To make all public and protected predicates declared via an extended object become private, we write:

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

Omitting the scope keyword is equivalent to writing:

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

Hence, by default, the scope of an object extension relation is public. The same rule applies to object instantiation and object specialization relations. This makes it possible to use a simplified syntax when protected or private inheritance is not necessary.

Finding the scope of object relations

The Logtalk built-in predicates presented in Chapter 1 for query the system about object relations are simplified versions of extended built-in predicates that also return the relation scope.

The built-in predicate `extends_object/3` returns prototype extension relations:

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

The built-in predicate `instantiates_class/3` returns object instantiation relations:

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

The built-in predicate `specializes_class/3` returns class specialization relations:

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

The simplified version of these predicates simply ignore the relation scope, returning all public, protected, and private relations.

3.3.2 Overriding inherited predicate definitions

When we define a predicate for which a definition is already inherited from an ancestor object, the inherited definition is hidden by the new definition. This is called overriding inheritance: a local definition overrides any inherited one. Assume, for example, we

have a class representing typical data about a person:

```
:- object(person,
    instantiates(metaperson),
    specializes(object)).

    :- public(age/1, height/1, weight/1).

    age(25).        % default age value
    height(170).    % default height value
    weight(75).     % default weight value

:- end_object.
```

This class provides default values for a person attributes such as `height/1`, `weight/1`, and `age/1`. Assume that we have the following class instance:

```
:- object(paul,
    instantiates(person)).

    age(37).
    height(175).

:- end_object.
```

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

```
| ?- paul::(age(Age), weight(Weight)).

Age = 37
Weight = 75
yes
```

Some Prolog object-oriented extensions such as SICStus Objects implement different behavior, where a new definition does not override the inherited ones. Instead, the new definition will be an additional solution to the corresponding message. For the example above, this design choice is clearly unwanted, but we can also find examples where the Logtalk implementation fails to meet the desired results. It may be argued that the Logtalk choice is more akin to common object-oriented languages, where extensions like as SICStus Objects are more closer to the Prolog notion of multiple, alternative solutions. However, the SICStus Objects behavior and other possible behaviors are easy to code in Logtalk, using the *super* control construct, as explained bellow.

3.3.3 Specializing inherited predicate definitions

An inherited predicate can be specialized by calling it in the new definition. This is accomplished by calling the `^^/1` operator in the new definition. A common example is a hierarchy of objects where each object defines some initialization code. Each object must perform, in addition to its own initialization code, any initializations inherited from its ancestor objects. Assume, for example, that we have the following prototype:

```
:- object(root).

    :- public(init/0).

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

:- end_object.
```

The definition of the predicate `init/0` in a descendant prototype must call the inherited definition:

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

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

:- end_object.
```

Sending the message `init/0` to the object `descendant` results in the following output:

```
| ?- descendant::init.

descendant init
root init
yes
```

This is the basic form of predicate specialization. We can also use the `^^/1` operator to code other interesting forms of specialization, as described next.

Union inheritance

Union inheritance is a variant of predicate specialization where all the definitions, both new and inherited ones, are taken into account. This is accomplished by writing a clause whose body contains only one call to the inherited definition (using the `^^/1` operator). The relative position of this clause, among the remaining clauses, sets the calling order for the local and inherited definitions. Consider the following example:

```
:- 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

Selective inheritance, sometimes also named differential inheritance [23], is a variant of predicate specialization where we hide some of the inherited definitions. This form of inheritance can be 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.

A typical example, adapted from L&O system cited above, is to define objects that represent birds and penguins. In this example, we simplified the descriptions of these classes of animals by restricting ourselves to the locomotion modes. For birds, we can define the following object:

```
:- object(bird).

    :- public(mode/1).

    mode(walks).
    mode(flies).

:- end_object.
```

Penguins are also birds, but they are unable to fly so we must exclude that locomotion mode:

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

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

:- end_object.
```

We can test our objects by calling the following goal:

```
| ?- penguin::mode(Mode).

Mode = swims ;
Mode = walks ;
no
```

3.4 Definite clause grammars

Definite clause grammar rules [72] 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 [73] is working on a draft specification [74] 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` 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`. Nevertheless, in case we want to call grammar rules the same way we call any other predicate, the corresponding predicate arity will be the arity of the rule head plus two. For the above example, assuming that we want the predicate corresponding to the `expr/1` rule to be public, the declaration would be:

```
:- public(expr/3).
```

In the body of a grammar rule, we can call rules that are inherited from ancestor objects (or imported 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. An example of composing grammar rules split in categories will be presented in Chapter 5. Along with the message sending operators (`::/1` and `::/2`), we can also use the control constructs `\+/1`, `!/0`, `;/2`, `->/2`, and `{}/1` in the body of a grammar. In addition, grammar rules may contain metacalls (a variable taking the place of a non-terminal), which are translated to calls of the built-in method `phrase/3`.

3.5 Built-in methods

Logtalk defines a set of built-in object predicates or built-in methods for accessing message execution context, finding sets of solutions, inspecting object predicates, and object database handling. Every Logtalk built-in method checks the type and mode of its calling arguments, throwing an exception in case of misuse. Similar to ISO Prolog built-in predicates, these built-in methods cannot be redefined. An exception will also be thrown if we attempt to redefine a Logtalk built-in method inside an entity. This section contains a brief description of these methods. The full specification of each method is contained in the Appendix B. Most built-in methods are based on the predicates specified in the ISO Prolog standards with the same name, contributing to a smooth learning curve for Prolog programmers.

3.5.1 Execution context methods

Logtalk defines four local built-in methods for accessing a method execution context and an object parameter values. A local built-in method can only be called directly, i.e., it cannot be used as a message to other objects (including *self*). The Logtalk pre-processor translates the calls of these built-in execution context methods to simple variable unifications that are performed at compilation time. We can thus use these methods freely without worrying about performance penalties.

To find the object that has received the current message we may use the method `self/1`. We may also retrieve the object that has sent the current message by using the method `sender/1`. The method `this/1` allows us to retrieve the name of the object containing the code being executed, instead of hard-coding the object name in a predicate definition. This helps to avoid bugs in our code, should we later decide to change the object name and forget to change the name references.

Here is a simple example of how to use these object execution context methods, consisting of two simple objects: a root object and a descendant object. The root object declares and defines a predicate, `test/0`, which calls the built-in context methods:

```
:- object(root).

    :- public(test/0).

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

:- end_object.
```

The descendant object simply extends the root object:

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

:- end_object.
```

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

```
| ?- descendant::test.

Executing a predicate definition stored in root
to answer a message received by descendant
that was sent by user
yes
```

For parametric objects, the method `parameter/2` enables us to retrieve current parameter values. The first argument is the parameter position in the object identifier (a compound term), while the second argument is the parameter value. 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 this object, we can try the following goal:

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

Color parameter value is blue
yes
```

Note that we may also use the method `this/1` to retrieve parameter values. For the example above, the definition of the `test/0` predicate could be rewritten as:

```
test :-
    this(block(Color)),
    write('Color parameter value is '),
    writeq(Color), nl.
```

However, this solution is not recommended because it implies that we have to hard-code the name of the object as the argument of the `this/1` call. The same reason has leaded us to include this built-in method in the first place.

3.5.2 Database methods

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

Static versus dynamic predicate declarations

In the following sections, I will need to make a distinction between a *static predicate declaration* and a *dynamic predicate declaration*. A static predicate declaration is a declaration that exists at entity creation time. That is to say, a declaration contained either in an object source file or in the arguments of the object creation built-in predicate, `create_object/4`, introduced in Chapter 1. A dynamic predicate declaration is a declaration that is (automatically) generated when we assert, at runtime, clauses for a new predicate into an object. Note that the concepts of static and dynamic predicate declarations are orthogonal to the concepts of static and dynamic objects. Static predicate declarations cannot be abolished during the lifespan of the container object. This is a conservative restriction intended to avoid problems with other objects defining or calling the predicate. Dynamic predicate declarations can be freely created and abolished at runtime.

Asserting predicate clauses

Predicate clauses can be asserted into an object using the built-in methods `asserta/1` and `assertz/1`. Similar to the Prolog built-in predicates with the same name, the first method asserts a clause as the first one for the predicate, while the second method asserts the clause as the last one for the predicate. In both cases, when the message is valid, the new clause is asserted in the object receiving the message. The asserting message is valid either if the asserted predicate is not declared for the object, or if it is declared as dynamic and within the scope of the *sender*.

Let me illustrate the possible cases using the `assertz/1` built-in method. The same rules do apply to the `asserta/1` method. A direct call of the `assertz/1` method such as:

```
Predicate :-
    ...,
    assertz(Clause),
    ... .
```

asserts a clause in the database of the object containing the predicate definition which makes the call, *this*. If the predicate has already been declared, it must have been declared as dynamic, otherwise an error will be thrown. If the predicate has not been yet declared for *this*, then a dynamic predicate declaration, with scope private, will be generated. If we need that the new predicate be declared public but still assert the clause in *this*, we can write instead:

```
Predicate :-
    ...,
    this(This),
    This::assertz(Clause),
    ... .
```

We can also send the `assertz/1` message to *self*:

```
Predicate :-
    ...,
    ::assertz(Clause),
    ... .
```

If the predicate has already been declared for *self*, then the predicate must have been declared as dynamic and within the scope of the *sender* of the message. That is, the predicate must be either public or protected. The predicate can also be private if it is declared in the *sender* of the message.

If the predicate has not yet been declared for *self*, then a dynamic predicate declaration, with scope protected, will be generated. If we need the new predicate to be declared public, but still assert the new clause in *self*, we can write, instead:

```
Predicate :-
    ...,
    self(Self),
    Self::assertz(Clause),
    ... .
```

Sending an `assertz/1` message to an object (other than *this* or *self*) asserts a new clause on the object if the corresponding predicate has not been yet declared yet or if it has already been declared as public and dynamic:

```
| ?- Object::assertz(Clause).
```

If the predicate has not been yet declared for the object, then a dynamic predicate declaration, with public scope, will be generated. When the predicate is declared protected or private, or is alternatively, public, but declared static, then an error will be thrown.

It is important to note that asserting a clause for a new predicate into an object does not imply that we can send the corresponding message to the object. That is only valid for prototypes but not for classes and instances due to the inheritance rules of class-based hierarchies. Assume, for example, that we have compiled and loaded the following two objects, a class and a class instance:

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

:- end_object.

:- object(instance,
    instantiates(class)).

:- end_object.
```

Assuming that `metaclass` contains no predicate declarations, let us now add a new predicate, `p/1`, to `class`, using the `assertz/1` built-in method:

```
| ?- class::assertz(p(1)).

yes
```

However, if we try to send the message `p/1` to `class`, an exception will be thrown:

```
| ?- class::p(X).

! error(
    existence_error(predicate_declaration, p(_)),
    class::p(_),
    user)
```

For `class` to understand the message `p/1`, the corresponding predicate must have been declared in its class, `metaclass`, or in a `metaclass` superclass. The message is only valid for the descendant instances of `class`:

```
| ?- instance::p(X).

X = 1
yes
```

Retracting predicate clauses

Logtalk provides two built-in methods for retracting predicate clauses: `retract/1` and `retractall/1`. The first method is similar to the predicate defined in the ISO Prolog standard with the same name. The second method has no corresponding predicate in the standard, but it is similar to a predicate available in most Prolog compilers with the same name. It allows us to retract all clauses whose head matches the method argument. Both methods operate on the database of the object receiving the corresponding messages. The same scope rules described for the asserting built-in methods do apply here: the

predicate for which we want to retract clauses must be within the scope of the *sender* of the retracting message.

The ISO Prolog standard states that retracting all clauses for a dynamic predicate does not abolish the predicate. Logtalk has adopted this rule, but the inheritance mechanism between objects implies some non-obvious consequences that I am going to illustrate through an example. Assume that we have compiled the following two objects:

```
:- object(root).

    :- public(p/1).
    :- dynamic(p/1).

    p(root).

:- end_object.

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

:- end_object.
```

Sending the message `p/1` to the object `descendant` returns the value inherited from the parent prototype, as expected:

```
| ?- descendant::p(Value).

Value = root
yes
```

Asserting a clause for the predicate `p/1` in `descendant` overrides the inherited definition:

```
| ?- descendant::(assertz(p(descendant)), p(Value)).

Value = descendant
yes
```

After retracting all clauses for the `p/1` predicate from `descendant`, the message `p/1` returns, once again, the inherited value as could be expected:

```
| ?- descendant::(retractall(p(_)), p(Value)).

Value = root
yes
```

For those with an object-oriented background, this is the expected outcome: looking up a method to answer the message `p/1` will lead us to the definition in `root`. However, another interpretation is possible when we apply the closed-world assumption to the local database of the object `descendant`: after retracting the definition of `p/1` from `descendant`, the predicate should fail in subsequent calls[2]. In the Logtalk design, the first interpretation was chosen, as illustrated in the example above. Besides, we can

[2]Under the closed-world assumption, everything that is not declared true, is false.

always assert the clause:

```
p(_) :-
    fail.
```

in the object `descendant` if the second interpretation should be necessary.

Abolishing predicates

As in Prolog, only dynamic predicates can be abolished. In Logtalk, abolishing a predicate implies abolishing its declaration (represented in source code by a scope directive). The same scope rules that we mentioned before, when describing the asserting and retracting built-in methods, apply to the built-in method `abolish/1`: the dynamic predicate that we wish to abolish must be within the scope of the object sending the message. In addition, the predicate must be declared on the object receiving the abolishing message.

However, because a predicate can be shared among several objects via inheritance, some restrictions are necessary in order to avoid inconsistency problems such as abolishing a predicate that is defined in descendant objects. In the first place, we can only abolish dynamically declared predicates. Secondly, we cannot abolish predicates declared in protocols or categories. The reason is simple: the same protocol may be implemented by several objects, and the same category may be imported by several objects. Besides, we can only send messages to objects.

3.5.3 Reflection methods

In other object-oriented languages such as Java, methods which enable us to query about an object methods and variables, are known as reflection methods. I have adopted the same terminology. Logtalk provides two built-in methods for inspecting object predicates. The first method, `current_predicate/1`, enables us to query about user predicate declarations. The second method, `predicate_property/2`, returns predicate properties. Both predicates were adapted, with some semantic changes, from the predicates with the same name specified in the ISO Prolog standards. These two predicates were defined in most Prolog compilers long before the standardization process, although there are some differences between implementations. The DECsystem-10 Prolog [61] defined a `current_predicate/2` predicate, while Quintus Prolog [36] introduced the predicate `predicate_property/2`.

There are two possible semantics for these reflection methods, corresponding to distinct points of view: the *database view* and the *protocol view*. In the database view, the reflection methods return information about the predicates declared inside an object. With this view, an object database may be inspected in the same way as a Prolog database. In the protocol view, the reflection methods return information about the visible predicates of an object, from the point-of-view of the *sender*. That is, the object interface or protocol. Both views return the same results for prototypes, but not for classes and instances. In fact, a prototype declares predicates for itself and for its descendants, while a class declares predicates for its instances but not for itself[3]. Logtalk implements the protocol view. It is interesting to note that the database view

[3]Not entirely true: a class may instantiate itself, as it often happens when programming class-based reflexive systems.

would reflect the Logtalk Prolog roots while the protocol view reflects its object-oriented nature.

Finding declared predicates

We can find all visible user-declared predicates, using backtracking, by calling the built-in method `current_predicate/1`. This method can be used to check whether a message to an object is valid or to enumerate an object protocol. Similar to the corresponding ISO Prolog predicate, all user predicates are found, whether they are static or dynamic. However, the ISO definition is extended to support and enforce predicate scope declarations. A direct call of this built-in method, without using the message sending operators, such as:

```
Predicate :-
    ...,
    current_predicate(Functor/Arity),
    ... .
```

returns all public, protected, and private predicates declared for the object.

If the message is sent from inside an object to *self*:

```
Predicate :-
    ...,
    ::current_predicate(Functor/Arity),
    ... .
```

then the call will return public and protected predicates declared for *self*, and private predicates declared in the *sender* for *self* (via inheritance), corresponding to the messages that we can send to *self*. Note that the private predicates of *self* itself are not returned because we are not sending the message from inside of it (unless, of course, *self* and *sender* are the same object).

If we send the message to an object other than *this* or *self*, as follows:

```
| ?- Object::current_predicate(Functor/Arity).
```

the call only returns public user predicates. This happens because we can only see the public interface from outside an object.

Note that the built-in method `current_predicate/1` returns an object interface as it is seen from the *sender* object, and not the predicates declared inside an object, matching the design decisions in the specification of the database handling methods. For prototypes, the method behavior is similar to the behavior of the predicate with the same name defined in the ISO standard for Prolog modules. However, the same is not true for classes and instances due to the different inheritance rules of prototype and class-based hierarchies.

Predicate properties

The properties of a visible predicate can be retrieved by sending to an object the message `predicate_property/2`. By design, this method enforces the predicate scope declarations. To enumerate the properties of a predicate visible in *this*, using backtracking, we

should call the method directly, without using the message sending operators:

```
Predicate :-
    ...
    predicate_property(foo(_), Property),
    ... .
```

To enumerate the properties of *self* predicates visible from the *sender*, using backtracking, we will write:

```
Predicate :-
    ...
    ::predicate_property(foo(_), Property)
    ... .
```

To enumerate the properties of a public predicate visible in an object (other than *self* or *this*), using backtracking, we will write:

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

The possible predicate property values are:

- `public`, `protected`, `private` — predicate scope
- `static`, `dynamic` — predicate compilation mode
- `built_in` — predefined predicate
- `metapredicate(Mode)` — metapredicate template
- `declared_in(Entity)` — entity containing the predicate scope directive
- `defined_in(Entity)` — entity containing the predicate definition that will be used to answer the corresponding message sent to the object that we are querying for the predicate properties

The properties `declared_in/1` and `defined_in/1` do not apply to built-in methods and Logtalk/Prolog built-in predicates. This set of properties is similar to the one defined in the ISO Prolog standard for modules, with an important difference: the properties `public`, `protected`, and `private` are scope properties and do not specify (as in the ISO standard) if a predicate definition can be retrieved by using the `clause/2` built-in predicate.

Note that calls such as the following one do not work:

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

because the first argument is not interpreted as a goal and this ISO Prolog specified built-in predicate knows nothing about Logtalk objects and messages.

Reflection built-in predicates in the ISO Prolog standard

The ISO standard for the Prolog module system specifies a `predicate_property/2` built-in predicate (section 7.2.2). Together with the `current_predicate/1` built-in predicate standardized in the first part of the ISO standard, they enable a Prolog programmer to perform some sorts of reflective computations. An annoying detail of the ISO definition of the `predicate_property/2` predicate is the poor choice of the atoms `public` and `private` to denote when a predicate definition can be retrieved using the built-in predicate `clause/2`. In most programming languages, including probably all object-oriented programming languages, a public procedure is one that can be called from outside its encapsulation unit, while a private procedure can only be called from inside. Using the same property names for denoting source code access will only cause trouble in the eventuality of a future revision that adds scope rules and scope predicate properties to the Prolog module system standard. Meanwhile, it will be a source of misunderstandings and a nuisance for those who come to Prolog with a background in object-oriented languages.

3.5.4 All solution methods

The usual meta-predicates for finding all solutions for a query are available in Logtalk as predefined methods: `bagof/3`, `findall/3`, and `setof/3`. There is also a `forall/2` method that implements a generate and test loop. These built-in methods can be used as follows:

```
| ?- Object::bagof(Term, Pred, List).
```

or, equivalently:

```
| ?- bagof(Term, Object::Pred, List).
```

Both goals give the same results. There is no advantage, other than code clarity, to prefer one form to the other.

3.5.5 Event handler methods

Logtalk support for event-based programming assumes that any object playing the role of a monitor defines two event handler methods: `before/3` and `after/3`. These methods will be discussed in Chapter 6.

3.5.6 Definite clause grammar parsing methods

As discussed in the previous section, Logtalk supports two definite clause grammar parsing built-in 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. At the time this thesis was written, Logtalk support for definite clause grammars was feature in active development. For example, the current implementation of the parsing methods only accepts as first argument a non-terminal instead of a rule body. Nevertheless, the Logtalk translator for definite clause grammar rules already outputs correct results for all test cases that have been tried, including some hard ones where some commercial Prolog compilers fail to give the expected translation, although the meaning of "correct translation" is still the source of some disagreement among the Prolog community.

3.6 Built-in predicates

Logtalk adds a new set of built-in predicates to those already specified in the ISO Prolog standard. This set includes predicates for:

- Enumerating objects, categories, and protocols
- Enumerating object, category, and protocol properties
- Enumerating object, category, and protocol relations
- Creating and abolishing objects, categories, and protocols
- Defining, enumerating, and abolishing events and monitors
- Compiling and loading objects, categories, and protocols
- Enumerating and setting compiler options

Some of these predicates have already been described in Chapter 1. The remaining built-in predicates will be introduced in the next chapters.

All Logtalk built-in predicates check the type and instantiation mode of the calling arguments, throwing an exception in case of misuse. All the exception terms thrown by Logtalk built-in predicates have the following format:

```
error(Error, Call)
```

For example:

```
error(type_error(object_identifier, 33), current_object(33))
```

A complete and detailed description of all Logtalk built-in predicates, including exception terms and conditions, can be found on appendix B.

3.7 Representing object state and behavior

In most object-oriented languages such as Smalltalk, Java, or C++ there is a strong distinction between state and behavior, variables and methods, or data and functions. This distinction does not exist in declarative languages such as Prolog and Logtalk, where we can view a set of clauses as both data and procedures. Nevertheless, we can always interpret an object predicate as representing either state or behavior. Very often, state is represented by predicate facts and behavior by predicate rules, but this is not in any way required by the Logtalk language. This section contains several examples that illustrate how to represent typical (and not so typical) concepts of common object-oriented languages through Logtalk object predicates. The idea is to provide a bridge between Logtalk and other object-oriented languages. However, I must emphasize that Logtalk is a declarative language and, as such, the use of asserting and retracting methods to implement object state should be minimized for the exact same reasons we should avoid using the asserting and retracting predicates in Prolog programming. Logtalk should be viewed as a declarative object-oriented language, adding code encapsulation and code reuse features to Prolog. Trying to apply to Logtalk programming the same kind of highly dynamic programming typical of other object-oriented languages, will result in poor performance programs.

3.7.1 Instance methods

Any predicate declared in a class works as an instance method, for all descendant instances. The predicate can be defined in the class, in the class subclasses, or in the instances themselves. This last option is not available in most object-oriented languages and will be described next.

Instance-defined methods

One of the advantages of prototypes over classes is the easy definition of objects with singular behavior. This is also partially available in the implementation of class instances in Logtalk. As for most class-based languages, methods must be declared in a class so that an instance will understand a message. In Logtalk, however, a method definition may be stored in an instance, thus overriding or specializing the class definition. The key for this functionality lies in the implementation of the *super* mechanism (represented by the `^^/1` control construct), which can be used in an instance-defined method to call an inherited definition in the instance class or in one of the superclasses of the instance class. In most object-oriented languages such as C++ or Java, an instance is little more than a glorified dynamic data structure representing the state of an object. From this point-of-view, instance-defined methods may seem a rather strange concept. However, this concept fits naturally in Logtalk where there is no "a priori" distinction between state and behavior, as both concepts are represented by predicates. Let me exemplify this concept with two applications of instance-defined methods.

Overriding and specializing methods in instances

I will start with a very simple example where we redefine and specialize an inherited method inside an instance. Assume that we have the following class:

```
:- object(my_class,
    instantiates(metaclass),    % metaclass
    specializes(object)).       % inheritance root

    :- public(method/0).

    method :-
        this(This),
        write('Method default definition, stored in '),
        writeq(This), write('.'), nl.

:- end_object.
```

We will now define three instances of `my_class`. The first instance, `instance1`, simply instantiates our class:

```
:- object(instance1,
    instantiates(my_class)).

:- end_object.
```

The second instance, `instance2`, overrides the method definition inherited from the class:

```
:- object(instance2,
    instantiates(my_class)).

    method :-
        this(This),
        write('This is an overriding definition stored in '),
        writeq(This), write('.'), nl.

:- end_object.
```

The third instance, `instance3`, specializes the method definition inherited from the class:

```
:- object(instance3,
    instantiates(my_class)).

    method :-
        this(This),
        write('Method specialized definition stored in '),
        writeq(This), write('.'), nl,
        write('Calls inherited definition using super:'), nl,
        ^^method.

:- end_object.
```

To test our instance-defined methods, we will now send the message `method/0` to each instance. For the object `instance1`, the definition of the method `method/0` is found in its class:

```
| ?- instance1::method.
Method default definition, stored in my_class.
yes
```

The object `instance2` overrides the inherited definition of the method `method/0`:

```
| ?- instance2::method.
This is an overriding definition stored in instance2.
yes
```

The object `instance3` specializes the inherited definition of the method `method/0`:

```
| ?- instance3::method.
Method specialized definition stored in instance3.
Calls inherited definition using super:
Method default definition, stored in my_class.
yes
```

Avoiding metaclass proliferation

Instance-defined methods are especially useful to avoid proliferation of metaclasses, whenever we just want to customize instantiation and initialization behavior for different classes. Assume, for example, that we want a class to keep a count of the number of instances created:

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

    % initialize instance counter at class load time...
    :- initialization(init_inst_counter).

    % call generic new/1 definition and increment counter
    new(Instance) :-
        ^^new(Instance),
        inc_inst_counter.

    init_inst_counter :-
        ...

    inc_inst_counter :-
        ...

:- end_object.
```

This code assumes that the instance creation method `new/1` is declared and defined in the class metaclass, `metaclass`, or in a metaclass superclass. This method is specialized in order to call a predicate that will increment the instance counter after successful creation of a new instance.

3.7.2 Class methods

Some languages such as Objective-C, C++, and Java, share the concept of class methods. In these languages, class methods are invoked in the class itself, not in the class instances, and are used primarily for instance creation and destruction. This concept of class methods is a consequence of the lack of support for metaclasses on those languages. In other languages such as Smalltalk or Logtalk, which support metaclasses, class methods are simply the methods of the class when interpreted as an instance and, as such, are declared in its metaclass and in the metaclass superclasses.

To rewrite, in Logtalk, object-oriented code written in languages that do not support metaclasses, we must simply define a metaclass to hold the class methods of the original classes. Note that in Logtalk, unlike Smalltalk-80, a metaclass can be shared by several instances: nothing forces the use a metaclass hierarchy parallel to the class hierarchy.

3.7.3 Instance variables

In common object-oriented languages, instance variable values are stored in the instances and are accessed and modified by methods defined in the instance classes or superclasses. In some languages, such as Smalltalk, instance variables are always private, while in other

languages such as C++ or Java they are usually declared private to enforce data hiding. We can easily accomplish the same in Logtalk. Consider, for example, the following class defining typical data about a person such as name and age:

```
:- object(person,
    instantiates(metaclass),  % metaclass
    specializes(object)).     % inheritance root

    :- private(name_/1).   % name instance variable
    :- dynamic(name_/1).

    :- public(name/1).     % accessor method for name_/1
    :- public(set_name/1). % setter method for name_/1

    name(Name) :-
        ::name_(Name).     % returns value defined for self

    set_name(Name) :-
        ::retractall(name_(_)),  % deletes  value from self
        ::asserta(name_(Name)).  % stores new value in self

    % declarations and methods for the age_/1 variable
    ...

:- end_object.
```

The use of the message sending operator `::/1` in the definition of predicates `name/1` and `set_name/1` ensures that we always access and modify the definition of the predicate `name_/1` of the instance which receives the corresponding messages.

Default values for instance variables

In Logtalk, to define a default value for an instance variable, we add the corresponding predicate clause at the class (or subclass) level. For example, expanding the previous example to include methods and declarations to represent a person's age, we may write:

```
:- object(person,
    instantiates(metaclass),
    specializes(object)).

    :- private(age_/1). % age instance variable
    :- dynamic(age_/1).
    ...
    age_(32).            % default age value

    age(Age) :-
        ::age_(Age).    % returns value defined for self
    ...

:- end_object.
```

When the message `age/1` is sent to an instance of the class `person`, the search for a definition of the predicate `age_/1` will start at the instance. However, if there is no local definition, the search will proceed to the instance class and then to the class superclasses until the default definition will be found in the class `person`.

Shared instance variables

Shared instance variables are instance variables whose value is shared by all instances. The variable value is stored at class level. In some languages such as Smalltalk-80, shared instance variables are misleadingly named class variables.

Assume, for example, that we have a hierarchy of objects representing geometric figures such as squares and triangles. The attribute "number of sides" can be considered an instance attribute but it does not make sense to store it in every square instance. Using a shared instance variable, the number of sides can be stored only once in the class.

Shared instance variables are easily implemented in Logtalk. Any predicate declared and defined in a class is shared by all instances unless we provide an overridden definition in the instance itself. We just need to define any accessor or setter methods to work at the class level, as in the following example:

```
:- object(my_class,
    instantiates(metaclass),  % metaclass
    specializes(object)).     % inheritance root

    :- private(siv_/1).    % shared instance variable
    :- dynamic(siv_/1).

    :- public(siv/1).      % accessor method for siv_/1 var
    :- public(set_siv/1).  % setter method for siv_/1 var

    siv_(0).               % siv value is local to the class

    siv(Value) :-
        siv_(Value).

    set_siv(Value) :-
        retractall(siv_(_)),
        asserta(siv_(Value)).

:- end_object.
```

Now assume that we have three instances of the above class, named `i1`, `i2`, and `i3`. Access and modification of the shared instance variable can be performed from any instance. We can start by getting the value of the shared variable for each instance:

```
| ?- i1::siv(V1), i2::siv(V2), i3::siv(V3).

V1 = 0
V2 = 0
```

```
V3 = 0
yes
```

We will now change the value of the shared variable via instance `i1`:

```
| ?- i1::set_siv(1).

yes
```

Getting the value of the shared variable for the other instances welds:

```
| ?- i2::siv(V2), i3::siv(V3).

V2 = 1
V3 = 1
yes
```

3.7.4 Class variables

Similar to class methods, class variables are the instance variables of a class seen as an instance. Consequently, class variables are declared in the class metaclass and in the metaclass superclasses.

3.7.5 Property sharing versus value sharing

Logtalk behavior of object predicates directly supports the concepts of *property sharing* and *value sharing*, found in object-oriented prototype languages. With property sharing, both property and its value are shared among descendant prototypes. This is similar to the concept of shared instance variables in class-based languages. With value sharing, the property is shared but its value is only shared by the descendant prototypes that do not override it. We can, thus, have a property with a default value that can be shared by some descendants while others may define specific values for it.

This section example, adapted from [59], illustrates how to perform both types of sharing. In this example, we have a prototype named `joePerson`, containing general data about a person, Joe, such as his age, name, and address. This prototype has three descendants or viewpoints: `joeSportsman`, `joeEmployee`, and `joeChessPlayer`. Each descendant contains data related to a particular viewpoint. We use the Logtalk built-in database methods, such as `asserta/1` and `retract/1`, in the context of *this* to implement property sharing, and in the context of *self* to implement value sharing. We can start with the definition of `joePerson`:

```
:- object(joePerson).

    :- public(getOlder/0).

    :- public(age/1).
    :- dynamic(age/1).

    :- public(name/1).

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

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

    :- public(setScore/1).

    age(30).
    name('Joe Smith').
    score(0).

    % use property sharing for the age/1 predicate:
    getOlder :-
        retract(age(Old)),
        New is Old + 1,
        asserta(age(New)).

    % use value sharing for the score/1 predicate:
    setScore(Score) :-
        ::retractall(score(_)),
        ::asserta(score(Score)).

:- end_object.
```

In this prototype, we used property sharing for the predicate `age/1` as all viewpoints refer to the same person. However, for the predicate `score/1` we used value sharing because this predicate will have a different interpretation, and thus a different value, for each viewpoint.

The prototype `joeSportsman` represents our view of Joe as a sportsman:

```
:- object(joeSportsman,
    extends(joePerson)).

    :- public(sport/1).
    :- public(stamina/1).
    :- public(weight/1).

    sport(snowboard).
    stamina(30).
    weight(111).

:- end_object.
```

Our view of Joe as an employee is represented by the prototype `joeEmployee`:

```
:- object(joeEmployee,
    extends(joePerson)).

    :- public(worksFor/1).

    :- public(salary/1).
    :- dynamic(salary/1).
```

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

    worksFor('ToonTown').
    salary(1500).

    % use property sharing for the giveRaise/1 predicate:
    giveRaise(Raise) :-
        retract(salary(Old)),
        New is Old + Raise,
        asserta(salary(New)).

:- end_object.
```

Joe is also a chess player:

```
:- object(joeChessPlayer,
    extends(joePerson)).

    :- public(category/1).

    category('National Master').

:- end_object.
```

After compiling and loading the objects above, we can ask Joe his age:

```
| ?- joePerson::age(Age).

Age = 30
yes
```

The same question could be made through any of his viewpoints, resulting in the same answer:

```
| ?- joeSportsman::age(Age).

Age = 30
yes
```

Now let us tell Joe to get older:

```
| ?- joePerson::getOlder.

yes
```

We can check the results of the above message from any of Joe's viewpoints. For example:

```
| ?- joeChessPlayer::age(Age).

Age = 31
yes
```

Because the `getOlder/0` updates the `age/1` predicate using property sharing, we can also send the `getOlder/0` message to any of the Joe's viewpoints, with the same results:

```
| ?- joeEmployee::getOlder.

yes
```

We can check this by asking Joe about its age again:

```
| ?- joePerson::age(Age).

Age = 32
yes
```

As you can see, although the update message has been sent to a descendant prototype, it is the predicate `age/1` in the parent prototype that was updated. To illustrate value sharing, we will use the predicates `score/1` and `setScore/1` defined in `joePerson`. We can start by retrieving the default value:

```
| ?- joePerson::score(Score).

Score = 0
yes
```

Initially, `counter/1` is only defined for `joePerson`, so, every descendant prototype, or viewpoint, will share its value or definition. For example:

```
| ?- joeEmployee::score(Score).

Score = 0
yes
```

However, we can set the score to some other value by sending the `setScore/1` message to a descendant. For example, the score for the national master chess category is 2200:

```
| ?- joeChessPlayer::(setScore(2200), score(Score)).

Score = 2200
yes
```

This viewpoint has now a local definition for `score/1`, that is independent of its parent (`joePerson`) definition:

```
| ?- joePerson::score(Score).

Score = 0
yes
```

All the other descendant prototypes or viewpoints continue to share the definition in the parent prototype `joePerson`:

```
| ?- joeSportsman::score(Score).

Score = 0
yes
```

3.8 Summary

Logtalk object predicates enable both a *logical view* and a *classical object-oriented view* of an object. That is to say, we can view an object predicate as representing something that is true about an object or as representing either object state or object behavior.

Data hiding is accomplished by declaring each predicate as public, protected, or private. The message sending mechanism enforces the scope rules, preventing any access to predicates outside the scope of the object sending the message. Logtalk also implements public, protected, and private inheritance, allowing us to restrict the scope of inherited predicates.

Object predicates can be used, together with the database built-in methods that allow us to assert and retract predicate definitions, to easily implement not only common concepts of object-oriented languages such as class and instance methods and variables, shared instance variables, and property and value sharing, but also not so common concepts such as instance-defined methods.

The support for dynamic object predicates allows us to modify, add, or remove state and behavior from an object at runtime. We can thus update whole object hierarchies by updating its root objects. All changes are immediately visible in descendant objects thanks to the use of dynamic binding in message processing and the independent compilation of each object.

A set of built-in object methods provides access to the execution context of message processing, complementing other features of the Logtalk language that allow the construction of applications performing all sorts of reflective computations. Logtalk also includes built-in object methods for database handling, reflection, and for finding all solutions to a query. These methods, being based on the built-in predicates with the same name defined in the ISO Prolog standards, enable a smooth transition from plain Prolog programming to Logtalk programming. In fact, an object predicate database implements most, if not all, of the functionality of Prolog's flat predicate database.

Chapter 4

Protocols

This chapter begins by presenting the Logtalk concept of protocol, comparing it to the concept of interface found on object-oriented languages and the ISO Prolog module system. Secondly, it explains how to define protocols and protocol hierarchies. Next, it describes the protocol directives and the built-in predicates for protocol handling. Finally, it specifies the Logtalk built-in reflection predicates for working with protocols.

4.1 Logtalk protocol concept

Logtalk protocols encapsulate predicate declarations. Ideally, the declarations should correspond to a functionally cohesive set of predicates. Protocols enable the separation between interface and implementation: a protocol can be implemented by several objects, and an object can implement several protocols. Note that, although abstract classes and multiple inheritance, both features supported by Logtalk, can provide some of the functionality of protocols, this solution can only be applied to class-based hierarchies. In contrast, Logtalk protocols can be implemented by both classes and prototypes.

An object implements a protocol by providing definitions for the declared predicates. However, the implementation of a protocol by an object should be interpreted in a loose sense. An object is not required to provide a definition for every protocol declared predicate. As discussed in Chapter 2, a message to an object is valid if the corresponding predicate is declared for the object and in the scope of the *sender*. If no predicate definition is found to answer the message, it simply fails. This semantics can be seen as a Logtalk reinterpretation of the Prolog *closed-world assumption*.

4.1.1 Related work

Logtalk concept of protocol is inspired by the concept of interface found in the Java object-oriented language. In addition, Logtalk protocols are designed to overcome the limitations of the current ISO standard for the Prolog module system.

Interfaces in object-oriented languages

A noteworthy difference between Logtalk and other object-oriented languages has to do with the rules for object implementation of a protocol. In languages such as Java, the failure to implement a method declared in an interface results in a compilation error.

Java interfaces Logtalk protocols subsume the functionality provided by Java interfaces. An interesting difference is that a Java interface can only contain declarations for public methods, while a Logtalk protocol may contain declarations for public, protected, and private predicates. The Java design reflects the notion of an object interface as its set of public methods. However, an object also provides a second interface for its (hierarchical) descendants. This interface includes both public and protected methods. In Logtalk, both interfaces can be declared using protocols. Similar to Java, Logtalk supports the definition of protocol hierarchies, which will be described later on this chapter.

Module interfaces in the ISO Prolog standard

The ISO Prolog module system standard allows us to separate interface form implementation. Modules are composed by a single module interface and zero or more module bodies. However, the identifier of a module interface must be the same of its module bodies, precluding the building of modules that implement more than one interface. In addition, it only allows a single implementation per interface. This rather limits the usefulness of modules. Often, applications use multiple implementations of the same interface. Modules implementing several interfaces are also common. These are features of any modern object-oriented language supporting the definition of independent interfaces, but their functionality is cumbersome to accomplish within this standard[1].

Interfaces in Prolog object-oriented extensions

The O'CIAO [43] object-oriented extension to CIAO Prolog [44] supports the definition of interfaces (as mentioned briefly in Chapter 1), based on the Java concept of interface. Unlike Logtalk, an object implementing an interface must provide a definition for all public predicates, otherwise an error will be thrown. In addition, O'CIAO only allow the declaration of public predicates. A class may also be used as an interface. In this case, all code in the class except the declarations of public predicates is ignored.

4.2 Working with protocols

This section describes the syntax for defining protocols and protocol hierarchies, the Logtalk built-in predicates for protocol handling, and the available protocol directives. The full specification of the directives and built-in predicates can be consulted on Appendix B.

4.2.1 Defining a new protocol

We can define a new protocol in the same way we can write Prolog code and define Logtalk objects: by using a text editor. Protocol names must be atoms. Protocols share a single namespace with objects and categories (presented in Chapter 5). As such, we cannot have a protocol with the same name as an object. Protocol directives are textually encapsulated by using two Logtalk directives: an opening directive, which can

[1] Partial workarounds are possible, at the expense of code clarity, by using predicate export, reexport, and import directives. However, as module interfaces cannot contain import directives, we still cannot define an interface as an extension of other interfaces.

be either `protocol/1` or `protocol/2`, and a closing directive, `end_protocol/0`. The simplest protocol will be one that is self-contained, not depending on any other Logtalk entity:

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

As an example, consider the following protocol, `listp`, containing declarations for common list processing predicates[2]:

```
:- protocol(listp).

    :- info([
        version is 1.0,
        author is 'Paulo Moura',
        date is 2000/7/24,
        comment is 'List processing protocol.']).

    :- public(append/3).
    :- mode(append(?list, ?list, ?list), zero_or_more).
    :- info(append/3, [
        comment is 'Appends two lists.',
        argnames is ['List1', 'List2', 'List']]).

    :- public(member/2).
    :- mode(member(?term, ?list), zero_or_more).
    :- info(member/2, [
        comment is 'Element is a list member.',
        argnames is ['Element', 'List']]).
    ...

:- end_protocol.
```

In most cases, protocols should be named after the functionality of their declared predicates. For example, a set of declarations for debugging predicates would be encapsulated in a protocol named `debugging`. Declarations for predicates handling monitors and events would be contained in a `monitoring` protocol. This, of course, is not always possible or desirable. In that case, it is common practice to append the character "`p`" to the name of (one of) the object(s) that will implement the protocol, as in the above example.

4.2.2 Protocol hierarchies

When a protocol extends one or more than one protocol, the opening directive will be:

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

[2]The directives `info/1` and `info/2` used in this example are discussed in Chapter 7.

In this case, `Protocol` will inherit all the declarations contained in the extended protocols. If `Protocol` declares a predicate that has already been declared in some extended protocol, the local declaration will override the inherited one.

Protocol hierarchies are useful when we want to define both a basic protocol and an extended version of the same protocol. Some objects may implement only the basic protocol while other objects may implement the extended protocol. As an example, which will be familiar to all Prolog programmers, assume that we define a basic protocol for debugging our programs through tracing of a goal execution:

```
:- protocol(tracing).

    :- public(trace/0).
    :- public(notrace/0).

:- end_protocol.
```

An object may implement this protocol in order to provide the user with a very basic debugger. However, for complex queries, tracing can be too verbose. Most of the time, we are more interested in spying the execution of a few specific predicates. We can then define a more sophisticated debugging protocol as an extension of the basic tracing protocol:

```
:- protocol(debugging,
    extends(tracing)).

    :- public(debug/0).
    :- public(nodebug/0).
    :- public(debugging/0).
    :- public(spy/1).
    :- public(nospy/1).
    :- public(nospyall/0).

:- end_protocol.
```

An object implementing this extended debugging protocol will also implement the basic tracing protocol. From the object point-of-view, it is as the protocol hierarchy had been flattened, with all predicate declarations collected in the implemented protocol declared in the object-opening directive.

There is another scenario where protocol extension is handy. A well-designed protocol should be both minimal and complete. However, for performance and practical reasons, we may want to extend a protocol with some convenient predicates that would make programming easier. Assume, for example, that we have defined the following protocol, `pointp`, containing basic predicates for representing two-dimensional geometric points using Cartesian coordinates:

```
:- protocol(pointp).

    :- public(x/1, y/1).
    :- public(set_x/1, set_y).

:- end_protocol.
```

Common point operations such as converting between Cartesian and polar coordinates, moving a point to a new location, or calculating the distance from a point to the origin, can be implemented by calling the methods declared in the above protocol. However, our code would be simpler if we define an extended protocol containing some convenient methods for these common operations:

```
:- protocol(extdpointp,
    extends(pointp)).

    :- public(move/2).
    :- public(ro/1, teta/1).
    :- public(distance/1).
    ...

:- end_protocol.
```

Without support for protocol hierarchies, we would be forced either to define a single protocol with all the above predicate declarations or to repeat some of the predicate declarations across different protocol versions. The first option would result in objects implementing predicates that might not be used. The second option would lead to code duplication between minimal and extended versions of the protocol.

4.2.3 Creating a new protocol at runtime

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

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

The first argument is the name of the new protocol (a Prolog atom). It must be different from other entity names. The second and third arguments correspond to the relations described in the opening protocol directive and to the protocol directives, respectively. For example, the following call:

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

is equivalent to compiling and loading the protocol:

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

    :- dynamic.

    :- public(foo/1).

:- end_protocol.
```

If we need to create many (dynamic) protocols at runtime, then it would be better to define an object containing a predicate that will call this built-in predicate in order to provide more sophisticated behavior.

4.2.4 Abolishing dynamic protocols

Dynamic protocols can be abolished at runtime by calling the Logtalk built-in predicate `abolish_protocol/1`:

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

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

4.2.5 Protocol directives

Protocol directives are used to set initialization goals, define protocol properties, and document protocols.

Protocol initialization

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

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

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

Dynamic protocols

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

```
:- dynamic.
```

Note that, as in most Prolog compilers, the performance of dynamic code is lower than the performance of static code. We should only use dynamic protocols whenever they will need to be abolished during program execution.

Protocol documentation

Similar to objects, a protocol can be documented with arbitrary user-defined information by using the `info/1` directive:

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

This directive will be fully discussed in Chapter 7.

4.2.6 Implementing protocols

An object can implement any number of protocols. The syntax is very simple:

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

The result is the same as if all protocol predicate declarations (that is, the predicate scope directives) were contained in the object itself. Therefore, when searching for a predicate declaration, implemented protocols (and the protocols that they may extend) are always searched before extended, instantiated, or specialized objects. In the case of the object and one of its implemented protocols declaring the same predicate, the local declaration will override the inherited one. In the event of two protocols declaring the same predicate, the declaration in the first protocol (following the order of the protocols in the `implements` clause) will override the declaration in the second protocol.

As an example, the following object implements the protocol `listp` presented earlier in this chapter:

```
:- object(list,
    implements(listp)).

    append([], List, List).
    append([Head| Tail], List, [Head| Tail2]) :-
        append(Tail, List, Tail2).

    member(Element, [Element| _]).
    member(Element, [_| List]) :-
        member(Element, List).

    ...

:- end_object.
```

Recall that, unlike the ISO Prolog modules, object and protocols must have distinct names since they share the same namespace.

We can also define alternative implementations of the protocol `listp` using, for example, difference lists:

```
:- object(difflist,
    implements(listp)).

    append(List1-Back1, Back1-Back2, List1-Back2).
    ...

:- end_object.
```

Public, protected, and private protocol implementation

Public, protected, and private protocol implementation is performed by prefixing, in the object-opening directive, the protocol name with the corresponding scope keyword.

To make all public predicates, declared via an implemented protocol, become protected, we write:

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

To make all public and protected predicates declared via an implemented protocol become private, we write:

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

Omitting the scope keyword is equivalent to writing:

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

Hence, by default, the scope of an implementation relation is public.

4.3 Finding about protocols

Logtalk provides a set of built-in predicates that allows us to perform reflective computations about protocols protocol relations in our programs.

4.3.1 Finding defined protocols

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

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

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

4.3.2 Protocol relations

Logtalk provides two sets of built-in predicates for querying the system about the possible relations between a protocol and other entities.

Protocol extension relations

The built-in predicate `extends_protocol/2` returns all pairs of protocols in which the first protocol extends the second one:

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

In case we also want to know the extension scope, we can use the built-in predicate `extends_protocol/3`, instead:

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

Protocol implementation relations

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

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

To also find the scope of the implementation relation we can use the built-in predicate `implements_protocol/3`, instead:

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

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

4.3.3 Protocol properties

A protocol may have the `static`, `dynamic`, or `built_in` property. We can find the properties of defined protocols by calling the built-in predicate `protocol_property/2`:

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

Dynamic protocols can be abolished at runtime by calling the `abolish_protocol/1` built-in predicate.

4.4 Summary

Traditionally, Prolog uses a single database for storing all predicates. Prolog modules change this through the implementation of namespaces, allowing us to split our database in more manageable parts. One problem with Prolog modules is that, although they support the concepts of module interface and module implementation, they fail to allow independent definition and use of interfaces and implementations, as found on modern object-oriented languages. Specifically, a module interface cannot be implemented by more than one module and a module cannot implement more than one interface. Logtalk protocols solve this problem by allowing several objects to implement a single protocol and an object to implement multiple protocols.

Logtalk supports the definition of protocol hierarchies. We can thus define a range of protocols from basic to sophisticate, covering from minimal to extended needs, without duplication of predicate declarations. In addition, Logtalk protocols may contain declarations for protected and private predicates, along with declarations for public predicates, allowing the definition of any kind of object interface.

Chapter 5

Categories

This chapter presents the Logtalk concept of *category*, a new composition mechanism which complements code reuse by inheritance and by object variable-based composition, especially in the context of single-inheritance languages. Firstly, it discusses the issues of code reuse in object-oriented languages. Secondly, it describes the concept of category, along with its roots and its implementation in Logtalk, followed by a comparison with related work. Next, it presents several examples. Finally, it summarizes some cases where categories are a useful tool and discusses some possible extensions of the category concept.

5.1 Code reusing

Sometimes we want to define and reuse a set of predicates that, even if functionally cohesive, do not fit in the notion of an object and only make sense when composed with other code in order to construct new objects. Take, for example, the Smalltalk [19] dependency mechanism that enables an object to notify a set of dependent objects of the occurrence of some relevant event. We may need this mechanism for some of our objects but certainly not for all. In Smalltalk, this is an all-or-nothing proposition. For example, in VisualWorks [75, 76] the code is contained in a parcel and is an optional feature that one can load/add to the base system. The dependent methods (and associated data structures) are always added to the root class, `Object`, making them available for all objects. We either have dependent support for all objects or for none. In other systems where the dependency code is included by default, it is also stored in the root class `Object`. There are two issues here: how do we encapsulate and how do we reuse such a set of methods? We seek a solution that enables us to add the methods that we want to reuse to the interface of only those objects that will use them, and at the same level as the object locally defined methods. In hybrid languages like C++ [8], we can always write generic code, such as utility functions, without encapsulating them inside objects. However, in pure object-oriented languages like Smalltalk or Java [9], all the code that we write must be encapsulated in some object. Sharing a method among unrelated objects will, then, require the use of either inheritance or composition mechanisms.

5.1.1 Inheritance-based reusing

While using inheritance, shared methods must be stored in a common ancestor. If the chosen language supports multi-inheritance, we can encapsulate our methods in

a parent class for all the objects that need to inherit such methods. Implementation multi-inheritance usually causes no problem for independent sets of cohesive methods and variables. However, languages like Smalltalk only support single-inheritance, while others such as Objective-C [21, 77] or Java only support the multi-inheritance of protocols. Without stepping into the single- versus multi-inheritance controversy [56], we need a solution that can be adopted by single-inheritance languages like Smalltalk or Java.

With single-inheritance, shared methods must be added to some common ancestor object. The outcome will often be root objects with large collections of methods and complex interfaces, despite the fact that most descendant objects never use most of the methods [22]. For example, the root class of VisualWorks 3.0 has 94 methods in 25 categories while Squeak 2.5 [78, 79], another Smalltalk system, has 96 methods in nine categories. The numbers for Java and Objective-C are better. In Java 1.2 [80] we have 12 methods in class `Object` versus 35 methods in class `Class`. In Apple's Objective-C frameworks [81] we have 38 instance and class methods in the root class `NSObject`. Single-inheritance may also force hierarchy relations that do not reflect the application domain but truly are workarounds for language limitations [82], although the problem can be mitigated by multi-inheritance of interfaces or protocols.

5.1.2 Object variable-based composition reusing

Common object-oriented languages like Smalltalk, Objective-C, Java, or C++, lack native support for composition at the same level as inheritance. The most common solution for implementing composition is to use an instance variable to hold a reference to an instance of the class that contains the methods we need to reuse. This implies a level of indirection whenever we want to add the composed object methods to the container interface, with the consequent need of cumbersome glue code and some performance penalties. In addition, updating the composed object will not automatically update the container object interface. Note that these drawbacks of what we call object variable-based composition result from our need of a different kind of composition solution, not from any inherent problem of this reusing method.

5.1.3 Category-based reusing

As an object-oriented language, Logtalk supports code reusing by inheritance and by object variable-based composition. However, Logtalk also provides an alternative form of code reuse through the concept of category, which will be presented next.

5.2 Logtalk category concept

The starting point for the Logtalk category concept comes from the Smalltalk-80 language where methods can be partitioned into named functional categories. However, Smalltalk-80 categories only have one documentation meaning. They are used to organize source code, and implemented by the language class browser. Logtalk extends this concept by turning a category into an encapsulation unit, at the same level as objects or protocols. The main idea is that we can compose a set of categories in order to define new objects, enabling code reuse without using inheritance or object-variable based composition. Conversely, any object may be split in a set of categories. The splitting

is straightforward and the code only requires elementary changes if predicates in one category need to call predicates in other category (because we are no longer calling code in the same encapsulation unit). The main purpose of categories is the encapsulation of functional sets of predicates, serving as object building blocks.

5.2.1 Category properties

Categories are fully implemented in the current Logtalk implementation, providing the following properties:

1. Categories have the same encapsulation power as objects: a category may contain both predicate directives and definitions. A category may also implement one or more protocols.

2. Category predicates are reused by importing the category into an object. The predicates are virtually added to the object protocol, along with any local object predicates, without any code duplication.

3. Categories provide runtime transparency: predicates added via a category are inherited by all the descendants of the importing object and can be called, redefined, or specialized like any other object predicate. One important consequence of this property is that an object can be factored in categories without breaking its clients or its descendants.

4. An object may import one or more categories. Any number of objects can import a category. A category is always shared between all importing objects with no duplication of code.

5. A category may declare and use dynamic predicates. In this case, each importing object will have its own set of clauses for each dynamic predicate. This enables a category to define and manage (object) state.

6. An object can restrict the scope of imported category predicates by prefixing the category name with one of the keywords public, protected, or private, in a similar way to public, protected, and private inheritance. By default, importation is public if the scope keyword is omitted.

7. Categories are compilation units; i.e., they are independently compiled from importing objects or implemented protocols, enabling incremental compilation.

8. There are no inheritance or importation mechanisms for categories. They cannot inherit from, or be inherited by, other categories or objects. They also cannot import, or be imported by, other categories. Thus it is both meaningless and an error to send a message to a category.

9. Categories enable an object to be virtually assembled only when created or loaded to memory. By importing one or more categories, an object will have a distributed dictionary of predicates composed of its own dictionary and of the dictionaries of each imported category. An object may then be updated simply by updating an imported category, without any need to recompile it or to access its source code.

10. Both classes and prototypes can import a category at the same time; its implementation is independent of the implementation of either type of objects. The use of categories is orthogonal to the choice of the most appropriate object concept, enabling the development of category libraries that can be reused in both prototype and class based designs.

11. Categories can be dynamically created and abolished at runtime (just like objects or protocols). However, note that runtime creation of new categories does not imply any kind of instantiation process: categories are not objects. Instead, Logtalk uses the same self-modifying code features found in Prolog.

Categories can be seen as a dual concept of Logtalk protocols: protocols provide interface reuse, while categories enable implementation reuse without using inheritance. Both protocols and categories are intended to encapsulate cohesive data. Both are used as building blocks in the definition of new, possibly unrelated, objects, allowing finer grain reuse. In addition, similar to a protocol, a category can be imported by several objects while an object can import several categories. However, where as protocols can extend other protocols, a category cannot be constructed as a composition of other categories. This can be seen as a limitation that constrains categories to be used as an enhanced virtual import mechanism, instead of a full blow separation of concerns or composition mechanism [83]. Nevertheless, despite its simplicity, categories enjoy several useful properties. But also because of its simplicity, categories are very easy to implement using current object compiling technology.

Conflicts may arise if two imported categories define the same predicate. This is akin to multi-inheritance conflicts but much simpler to spot and solve because categories do not inherit from other categories or objects. In addition, a category is mainly used to encapsulate a set of functionally cohesive predicates, thus minimizing the chances of name conflicts. The current Logtalk version uses a simple depth-first lookup when searching for a predicate, implicitly solving any possible name clashes. If an object inherits a predicate that is also defined in an imported category, the category definition takes precedence over the inherited definition (this is similar to overriding a method in a subclass). If a category defines the same predicate as an object into which it is imported, the object predicate overrides the predicate definition defined in the category.

5.2.2 Implementation

The current Logtalk version contains a full implementation of all the properties of the category concept, as described in this chapter. The system also includes several examples of the use of categories, some of them presented here in the next session. It should be noted that the Prolog/Logtalk features of interpreting code as both data and executable procedures, combined with easy conversion between data and code, make it arguably easier to implement features like categories when compared to languages like C++, Smalltalk, or Java. The Logtalk compiler and runtime source files can be downloaded from the Logtalk web site for close examination of the implementation details.

Objects that import categories and/or implement protocols have a distributed dictionary of predicates. That is to say that, in addition to a list of local predicates, objects have links to the dictionaries of imported categories and implemented protocols. These links, defined at compilation time, are always searched for before inheritance links. Categories (and protocols) are compiled such as the encapsulated code can be shared

and used by several objects (either prototypes or instances/classes) at the same time. This is accomplished in two steps. First, category predicates are compiled like object predicates, with extended arguments for the execution context information. This extended arguments include *self* (the object that received the original message), *this* (the object importing the category, that virtually contains the predicate under execution), and *sender* (the object that has sent the original message). Secondly, at runtime, the object–category dictionary links propagate the current execution context to the category predicates, enabling them to be used like they have been defined in the importing object. That is, when executing a definition contained in a category, the values of *self*, *this*, and *sender* are the same as if the definition was contained in the object importing the category. In the case of dynamic predicates (predicates whose definition can be modified at runtime), the implementation of the predefined methods that allow us to add, change, and delete definitions ensure that each importing object will have its own set of definitions.

Another important aspect is the performance cost of adding categories to an object-oriented language. If all imported categories only contain predicate directives, then performance should be similar to languages like Objective-C or Java that implement multi-inheritance of protocols. In the most common situation, where a category contains both predicate directives and definitions, searching for a predicate may require looking inside an imported category. This will have a small performance penalty that is proportional to the number of imported categories, and results from the need to access several encapsulation units.

As categories can provide alternative solutions to the use of multi-inheritance (see the points example in the next section), we should also compare the costs of these two reusing methods. While an inheritance link may lead to several other inheritance links, following an imported category link implies only one level of indirection when searching an importing object predicate dictionary: a category does not inherit or import code from other objects or categories. This ensures that a design using single-inheritance and categories has a more predictable and better method-lookup performance than an equivalent multi-inheritance solution.

5.3 Related work

In this section, some related work on other programming languages is discussed and compared with the Logtalk concept of category.

5.3.1 Mixins

The Flavors [84] system, an object-oriented extension to LISP [85], introduced the concept of *mixin* [86], a coding convention that uses abstract subclasses to specialize behavior in parent classes. Mixins are combined, using multi-inheritance, with other mixins to build regular classes. Mixins, like Logtalk categories, often encapsulate a set of functionally cohesive methods and attributes. However, categories are reused by composition while mixins are reused through multi-inheritance. In a language that supports multi-inheritance, mixins enable flexible reusing without the need of introducing a new kind of entity. Another important difference is that, while mixins rely on specialization of parent methods (using the call-next-method primitive), categories do not need to depend on importing objects. Categories are often used to encapsulate independent, self-contained

code, resulting in more flexible and powerful reusing mechanism. We can only use a mixin if the class that inherits the mixin also inherits a parent that defines the method specialized by the mixin. No such constraints exist in reusing Logtalk categories.

5.3.2 Smalltalk categories

Regarding Smalltalk, most implementations define interface primitives for loading and saving fragments of code. These primitives, historically named `FileIn` and `FileOut`, enable the programmer to add or remove fragments of code, methods and variables, from a class. These fragments of code often correspond to a category or a set of categories, thus giving a useful operational meaning to an otherwise documentation only concept. However, a Smalltalk category is always associated with a specific class and cannot be shared by two or more classes. Logtalk removes this restriction, generalizing the category concept to enable a category to be imported into any object.

5.3.3 Objective-C categories

The Objective-C language also implements a category concept, but as a way to extend an existing class with new methods, even when the extended class source code is not available. It can be seen as an alternative to a sub-class. However , one cannot extend a class other than the one specified in the category declaration. This differs from Logtalk, where categories are independent of objects, and any category can be imported by any object. While Objective-C categories are designed to extend existing code, Logtalk categories are object building blocks. Although two different concepts, aiming at different goals, they share some important properties such as run-time transparency, encapsulation of related methods, incremental compilation, and easier maintenance of complex objects.

5.3.4 Ruby modules

Ruby [87, 88, 89] is an object-oriented scripting language that supports a concept of *mix-in modules* similar to Logtalk categories. The programmers reference guide [90] defines modules as:

> "Ruby intentionally does not have the multiple inheritance as it is a source of confusion. Instead, Ruby has the ability to share implementations across the inheritance tree. This is often called 'Mix-in'."

Ruby modules are collections of methods and constants that can be imported by any class. Modules cannot be instantiated or subclassed but can receive messages. This means that we can use module resources without first importing them into a class. It also means that we must be careful in not calling methods that only make sense when doing mix-in programming.

5.3.5 Prototype languages

The representation of shared code (similar to what we can do with classes) has always been a problem in prototype-based languages. The Self [60, 91] prototype programming language defines a concept of *trait prototypes* that are used to store common behavior, playing a role similar to the role played by classes in class-based languages. One

drawback of traits is the fact that, in spite of being objects, they cannot answer most messages because the corresponding methods need access to slots only available in descendant prototypes [29]. Logtalk categories can play the role of traits as a way to store shared methods with the advantage that is not possible to send a message to a category.

As described in the previous section, Logtalk categories are an evolution of the Smalltalk-80 functional category concept, taking what is essentially a browser documentation feature and transforming it into a code reuse language mechanism. As such, it compares favorably to other reuse mechanisms at the same level as mixins, multi-inheritance, or object variable-based composition. However, it does not intend to compete with higher-level solutions for code reuse through composition such as aspect-oriented programming [92], subject-oriented programming [93], or binary component adaptation [94] among others.

5.4 Working with categories

This section describes the syntax for defining categories, the Logtalk built-in predicates for category handling, and the available category directives. The full specification of the directives and built-in predicates can be consulted on Appendix B.

5.4.1 Defining a new category

We can define a new category in the same way as we define objects and protocols and write Prolog code: by using a text editor. Category names must be atoms. Objects, categories, and protocols share a single name space. Thus, we cannot have a category with the same name as an object or a protocol. Category code (directives and predicates) is textually encapsulated between two Logtalk directives: `category/1-2` and `end_category/0`.

The simplest category will be one that is self-contained, not depending on any other Logtalk entity:

```
:- category(Category).

    ...

:- end_category.
```

When a category implements one or more protocols then the opening directive would be:

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

    ...

:- end_category.
```

Note that a category cannot import other categories or inherit code from an object. Nevertheless, in the same way as objects, categories support public, protected, and private protocol implementation.

5.4.2 Creating a new category at runtime

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

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

The first argument, the name of the new category (a Prolog atom) should not match with an existing entity name. The remaining three arguments correspond, respectively, to the relations described in the opening category directive, to the category directives, and to the category clauses. For example, the following 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 it would be better to define a metaclass or a prototype with a predicate that will call this built-in predicate in order to provide more sophisticated behavior.

5.4.3 Abolishing dynamic categories

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

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

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

5.4.4 Category directives

Category directives are used to set initialization goals, to define category properties, and for documenting categories.

Category initialization

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

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

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

Dynamic categories

As it 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.
```

Note that, as in most Prolog compilers, the performance of dynamic code is lower than the performance of static code. We should only use dynamic categories whenever they will need to be abolished during program execution.

Category dependencies

In addiction to the relations declared in the category-opening directive, the predicate definitions contained in the category may imply other dependencies. This can be documented by using the `calls/1` and the `uses/1` directives.

The `calls/1` directive can be used when a predicate definition sends a message that is declared in a specific protocol:

```
:- calls(Protocol).
```

If a predicate definition sends a message to a specific object, this dependence can be declared with the `uses/1` directive:

```
:- uses(Object).
```

These two directives can be used by the Logtalk runtime to ensure that all needed entities are loaded when running an application.

Category documentation

Similar to objects and protocols, a category can be documented with arbitrary user-defined information by using the `info/1` directive:

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

This directive will be fully discussed in Chapter 7.

5.4.5 Importing categories

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

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

The result is the same as if all category predicate declarations and definitions are contained in the object itself. When searching for a predicate declaration or definition, imported categories (and the protocols that they may implement) are always searched after implemented protocols, and before extended, instantiated, or specialized objects. In the case that the object declares (or defines) a predicate already declared (or defined) in a imported category, the local declaration will override the inherited one. In the event of two categories declaring (defining) the same predicate, the declaration (definition) on the first category (following the order of the categories in the `imports` clause) will override the declaration (definition) on the second category.

Public, protected, and private category importation

Public, protected, and private category importation is performed by prefixing, in the object-opening directive, the category name with the corresponding scope keyword.

To make all public predicates declared via an imported category become protected, we write:

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

To make all public and protected predicates declared via an imported category become private, we write:

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

Omitting the scope keyword is equivalent to writing:

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

Hence, by default, the scope of an importation relation is public.

5.4.6 Handling dynamic predicates

Categories cannot contain clauses for dynamic predicates. The major reason for this limitation is that a category can be imported by several different objects. While category static predicates can be shared among any number of objects without problems, sharing dynamic predicates would raise two problems. First, we cannot send messages to categories. Therefore, we cannot use built-in methods such as `assertz/1` or `clause/2` on clauses of dynamic predicates defined inside the category. These built-in methods always act on the database of the object that receives the corresponding message. Second, even if we could send messages to categories, it would not be possible to ensure that all updates of a dynamic predicate definition would be compatible with all the objects importing the category. However, there are no restrictions in declaring dynamic predicates or in defining predicates that handle dynamic predicates. For example, if we want to define a category containing predicates implementing variables using destructive assignment, we could write:

```
:- category(variable).

    :- public(get/2, 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.
```

Note that the dynamic predicate is called by the predicate `get/2` using the `::/1` message sending control construct. This will ensure that the correct predicate definition for each object importing the category will be used. In the same way, the predicate and `set/2` always update the correct definition, contained in the object receiving the corresponding messages `retractall/1` and `asserta/1`. This way, each object importing the category will have its own definition for the `value_/2` private predicate.

5.5 Finding about categories

Logtalk provides a set of built-in predicates that allows us to perform reflective computations about categories and category relations in our programs.

5.5.1 Finding defined categories

We can enumerate, using backtracking, all defined categories by calling the Logtalk built-in predicate `current_category/1` 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 (that is, an atom).

5.5.2 Category relations

Logtalk provides two sets of built-in predicates for querying the system about the possible relations between a category and other entities.

Implementation relations

To find out which categories implement which protocols we can use the built-in predicates `implements_protocol/2` and `implements_protocol/3`:

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

or, if we want to know the implementation scope:

```
| ?- implements_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.

Importation relations

To find out which objects import which categories we can use the built-in predicate `imports_category/2`:

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

or, if we want to know the importation scope, we can use instead the built-in predicate `imports_category/3`:

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

Note that several objects can import one category.

5.5.3 Category properties

A category may have the `static`, `dynamic`, or `built_in` property. We can find the properties of defined categories by calling the built-in predicate `category_property/2`:

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

Dynamic categories can be abolished in runtime by calling the `abolish_category/1` built-in predicate.

5.6 Examples

The following examples show how categories may be used for component-based programming, compare object variable-based with category-based composition, and illustrate how categories can be used for sharing code between selected objects, providing alternative solutions to multi-inheritance. Although there are no predefined categories in Logtalk, several examples can be found in the current Logtalk distribution, both in the standard library and in the tutorial examples.

5.6.1 Composing definite clause grammars

Definite clause grammar rules can be contained in categories, like regular predicates. This allows complex grammars to be modularized by splitting them in categories that can be combined and reused as necessary.

This example illustrates how to construct a simple grammar for parsing natural language sentences from a set of categories, each one containing a set of definite clause rules. The first category encapsulates definite clauses for determiners:

```
:- category(determiners).

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

:- end_category.
```

The second category contains rules for common nouns:

```
:- category(nouns).

    noun --> [boy].
    noun --> [girl].

:- end_category.
```

The third category contains rules for common verbs:

```
:- category(verbs).

    verb --> [likes].
    verb --> [hates].

:- end_category.
```

We can now compose these three categories by importing them into an object, which adds the necessary rules to define sentences through the combination of determiners, nouns, and verbs:

```
:- 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.
```

Note that the calls to the non-terminals `determiner`, `noun`, and `verb` are prefixed by the message sending operator `::/1` in order to call the grammar rules in the imported categories.

Compiling and loading the above object and categories enables queries such as the following ones:

```
| ?- sentence::parse([the, girl, likes, the, boy], Result).

Result = true
yes

| ?- sentence::parse([the, girl, scares, the, boy], Result).

Result = false
yes
```

The implementation of the built-in method `phrase/3` allows sentences to be both parsed and generated, depending naturally on the grammar rules that we define. For this example, this makes it possible to use queries such as follows:

```
?- sentence::parse(Sentence, true).

Sentence = [the, boy, likes] ;
Sentence = [the, boy, hates] ;
Sentence = [the, boy, likes, the, boy] ;
Sentence = [the, boy, likes, the, girl] ;
...
```

in order to generate all possible sentences recognized by the grammar.

5.6.2 Splitting an object in categories

Two of the most basic benefits of categories are code documentation and code organization. Most Smalltalk implementations already classify methods in several functional categories. In these cases, splitting a class using the Logtalk category concept is a trivial job (when a method in one category needs to call a method in another category, some elementary code changes will be needed). Let us turn our attention to Java instead. Take, for example, the class `Float`, contained in the `java.lang` package [80]. This class declares twenty-three new methods that can be easily classified in five categories named *constructors*, *comparing*, *testing*, *converting*, and *hashing* as follows:

```
constructors
    Float(double value)
    Float(float value)
```

```
    Float(String s)

comparing
    compareTo(Object o), compareTo(Float anotherFloat)
    equals(Object obj)

testing
    isNaN(), isNaN(float v)
    isInfinite(), isInfinite(float v)

converting
    toString(), toString(float f)
    parseFloat(String s), valueOf(String s)
    floatToIntBits(float value), intBitsToFloat(int bits)
    floatValue(), doubleValue()
    shortValue(), intValue(), longValue()
    byteValue()

hashing
    hashCode()
```

This helps the programmers in locating a method to perform a specific type of service while browsing a class, and results from the simple fact of classifying the methods in appropriate functional categories. The documentation of the Java classes present methods in alphabetical order, handy only if we are looking for the details of a known method. It should be noted however that, just as a class hierarchy implicitly reflects a specific classification point-of-view over a set of objects, there is usually more than one way to split a set of methods into functional categories.

5.6.3 Categories as a complementary composition tool

The category concept here presented provides a composition mechanism different from what we may call object variable-based composition. Usually, composition is accomplished by storing references to other objects in object variables. Comparing the two mechanisms shows they are complementary and addressing different needs, rather than competing ways of doing composition. Object variable-based composition is mainly used to implement part-of hierarchies. The implied level of indirection is often used in our advantage to control which methods (if any) are made available to the clients of the container object. By contrast, the methods imported from a category are transparently used and are conceptually at the same level of any object-defined method. Object variable-based composition also implies the creation of new objects every time a container object is instantiated and, therefore, a policy to control the process. It is however free from name clashes that may affect multi-inheritance or category-based composition solutions.

Let us start by defining a category that implements a set of predicates for handling a dictionary of attributes. We will need public predicates to set, get, and delete attributes, and a private dynamic predicate to store the dictionary entries. Let us name these predicates `set_attribute/2` and `get_attribute/2`, for getting and setting an attribute

value, `del_attribute/2` for deleting attributes, and `attr_/2`, for storing the attribute–value pairs:

```
:- category(attributes).

    :- public(set_attribute/2). % set a pair attr-value
    :- public(get_attribute/2). % test/get a pair attr-value
    :- public(del_attribute/2). % delete a pair attr-value

    :- private(attr_/2).        % attributes storage
    :- dynamic(attr_/2).

    set_attribute(Attr, Value):-
        ::retractall(attr_(Attr, _)),
        ::assertz(attr_(Attr, Value)).

    get_attribute(Attr, Value):-
        ::attr_(Attr, Value).

    del_attribute(Attr, Value):-
        ::retract(attr_(Attr, Value)).

:- end_category.
```

If necessary, we can put the predicate directives inside a protocol that will be implemented by the category:

```
:- category(attributes,
    implements(attributes_protocol)).
    ...
:- end_category.
```

We reuse the category predicates by importing them into an object:

```
:- object(person,
    imports(attributes)).
    ...
:- end_object.
```

After compiling and loading this object and our category, we can now try queries like:

```
| ?- person::(set_attribute(name, paul), set_attribute(age, 36)).

yes

| ?- person::get_attribute(Attribute, Value).

Attribute = name, Value = paul ;
Attribute = age, Value = 36 ;
no
```

Note that the interface of the category `attributes` is now part of the interface of the object `person`. Most object-oriented programming language libraries provide dictionary classes that we can reuse in our applications, either by multi-inheritance or by composition. In this example, multi-inheritance would result in viewing the object `person` as a kind of dictionary, hardly an elegant solution. With object variable-based composition, glue code will be needed to add the desired dictionary methods to the public interface of `person`. Category-based composition thus provides an alternative solution without any of these problems. Moreover, the resulting category can be reused in same application or in other applications.

To further illustrate the differences between object variable-based and category-based composition let me present another example using input/output operations. Assuming a stream-based input/output model, any object may need to define, redirect, open, read, write, or close new streams. Using a category-based approach, we may start by defining a streaming category, containing predicates to maintain a dictionary of currently defined streams:

```
:- category(streaming).

    :- public(stream/2).     % test/get pair name-stream
    :- public(set_stream/2). % define a new pair name-stream
    ...

:- end_category.
```

This category may also implement common stream operations. Any object whose interface must include stream input/output operations will, then, be able to import this category:

```
:- object(an_object,
    imports(streaming)).
    ...
:- end_object.
```

This way, it would be possible for us to query the object (or its descendants) about the streams it defines by using messages such as:

```
| ?- an_object::stream(Name, Stream).
```

If an object variable-based solution is preferred or needed, we can still reuse the streaming category by first importing the category into a class:

```
:- object(streams,
    imports(streaming),
    instantiates(class),    % some suitable metaclass
    specializes(object)).    % some suitable inheritance root
    ...
:- end_ object.
```

We can now store instances of this object in instance-variables of any object that needs to perform stream input/output operations. For example:

```
:- object(an_object,
    imports(attributes)).  % from the previous example

    init :-
        streams::new(Strs),
        ::set_attribute(streams, Strs),
        ...
    ...

:- end_ object.
```

However, the streaming methods can no longer be used directly:

```
| ?- an_object::set_stream(Name, Stream).

uncaught exception:
    error(existence_error(
        predicate_declaration, set_stream(Name, Stream)),
        an_object::set_stream(Name, Stream), user)
```

Messages like this will now generate unknown message exceptions because the streaming protocol is no longer part of the object protocol.

5.6.4 Hierarchy relations

One of the Logtalk companion examples defines a set of categories implementing methods for inspecting object hierarchy relations. Some methods can be defined for both prototype and class hierarchies, and their declarations can be abstracted in a common protocol:

```
:- protocol(hierarchyp).

    :- public(leaf/1).    % test/get an hierarchy leaf
    :- public(leaves/1).  % get list of all hierarchy leaves
    ...

:- end_protocol.
```

For prototype hierarchies, we can define methods such as `parent/1`, `ancestor/1`, or `descendant/1`:

```
:- category(p_hierarchy,
    implements(hierarchyp)).

    :- public(ancestor/1).   % test/get an ancestor prototype
    :- public(descendant/1). % test/get a descendant prototype
    :- public(parent/1).     % test/get a parent prototype
    ...

:- end_category.
```

While for instance/class relations we can have methods such as `instance/1`, `class/1`, or `superclass/1`:

```
:- category(ic_hierarchy,
    implements(hierarchyp)).

    :- public(class/1).      % test/get an instance class
    :- public(instance/1).   % test/get a class instance
    :- public(superclass/1). % test/get a class superclass
    ...

:- end_category.
```

Although these methods are potentially useful for any object, most objects will never use them. Most applications do not need to perform reflective computations. In languages like Smalltalk or Java, this kind of methods must be added to the root object in order to be available for any object that may need them. By encapsulating these methods in a category, they can be added to the interface of only those objects that really need them.

5.6.5 Monitoring category

Besides integrating logic and object-oriented programming, Logtalk also supports event-driven programming where an event is generated every time a message is exchanged between objects. Any object may act as a monitor for a registered event. A minimal monitor protocol consists only of a callback method, but more sophisticated behavior is possible. For instance, an object may need to keep a dictionary of events that can be modified, activated, or suspended. However, not all application objects will act as monitors and, among those that do, some may only need basic behavior. Encapsulating the monitor methods in a root object will ensure the requirement that any object may perform a monitor role but will also just clutter the interface of non-monitor objects. Moreover, not all applications use event-driven programming. Defining a category containing monitoring predicates will solve these problems easily:

```
:- category(monitoring).

    :- public(activate/0).   % start monitoring
    :- public(suspend/0).    % suspend monitoring

    :- public(reset/0).      % stop and delete all events

    :- public(add_event/4).  % define a new event
    :- public(del_event/4).  % delete a defined event
    :- public(event/4).      % test/get a defined event
    ...

:- end_category.
```

Any object that needs more complex monitor behavior just needs to import this category:

```
:- object(my_monitor,
    imports(monitoring)).
    ...
:- end_object.
```

Alternatively, the root of any sub-hierarchy of monitor objects may import the category. This way, objects that will never perform the role of monitors will not need to inherit a set of useless methods. Applications that do not use event-driven programming will not need to include code that will never be called.

5.6.6 Points

This example shows how categories may be used as an alternative to multi-inheritance solutions. The description of the original problem can be found in the SICStus Objects documentation [24], along with a solution using message delegation.

Assume that we want to represent points in a two-dimensional space. We can start by creating a point class defining a method `move/2` to translate a point to a new position, and a method `print/0` that outputs the current position:

```
:- object(point,
    instantiates(class),
    specializes(object)).

    :- public(move/2).      % move point to a new position
    :- public(position/2).  % test/get point position
    :- public(print/0).     % output current point position

    :- private(xy_/2).      % point position storage
    :- dynamic(xy_/2).

    move(X, Y) :-
        ::retractall(xy_(_, _)),
        ::assertz(xy_(X, Y)).

    position(X, Y) :-
        ::xy_(X, Y).

    print :-
        self(Self),
        ::xy_(X, Y),
        writeq(Self), write(' @ '), write((X, Y)), nl.

:- end_object.
```

From this base class, we would want to derive two sub-classes: `bd_point` and `hst_point`. Instances of `bd_point` can only move around in a restricted area. Instances of `hst_point` keep the history of its past positions. The new classes are easily defined by specialization of the `move/2` and `print/0` methods:

```
:- object(bd_point,
    instantiates(class),
    specializes(point)).

    :- private(bds_/3).     % coordinate bounds storage
    :- dynamic(bds_/3).

    move(X, Y) :-
        ::bds_(x, MinX, MaxX),
        X >= MinX, X =< MaxX,
        ::bds_(y, MinY, MaxY),
        Y >= MinY, Y =< MaxY,
        ^^move(X, Y).

    print :-
        ::bds_(x, MinX, MaxX),
        writeq(bds(x)), write(': '), write((MinX, MaxX)), nl,
        ::bds_(y, MinY, MaxY),
        writeq(bds(y)), write(': '), write((MinY, MaxY)), nl,
        ^^print.

:- end_object.
```

Similar for the `hst_point` class:

```
:- object(hst_point,
    instantiates(class),
    specializes(point)).

    :- private(hst_/1).     % position history storage
    :- dynamic(hst_/1).

    move(X, Y) :-
        ::position(X0, Y0),
        ^^move(X, Y),
        ::retract(hst_(Hst)),
        ::assertz(hst_([(X0,Y0)| Hst])).

    print :-
        ::hst_(Hst),
        write('history: '), write(Hst), nl.
        ^^print.

:- end_object.
```

Now, assume that we want to define another sub-class, named `bd_hst_point`, which combines the behavior of both `bd_point` and `hst_point`. This would suggest a multiple inheritance solution: `bd_hst_point` clearly specializes both `bd_point` and `hst_point`, sub-classes of `point`. However, this solution, even if possible, hides several problems.

The first obstacle would be that the bounded and the history behavior are embedded in the specialization of methods move/2 and print/0. Defining new methods such as `check_bds/2` and `print_bds/2` in class `bd_point`, and `add_to_hst/2` and `print_hst/0` in class `hst_point` can easily solve this particular problem. A bigger problem would be the fact that the basic behavior for moving or printing a point is defined in class `point`. However, being the corresponding methods redefined in classes `bd_point` and `hst_point`, how does one call the original definitions stored in point? Note that if the methods `move/2` and `print/0` are inherited from both `hst_point` and `bd_point`, a point will be then moved and printed twice. If the inheritance is carried out, for each method, only from one of the superclasses, then we will be breaking the problem symmetry. The class `bd_hst_point` could build its own definitions of methods `move/2` and `print/0`, by redefining the methods inherited from one superclass in order to call the methods specific of the other superclass. However, this solution would also prove to be problematic. Let us assume that the method `move/2` is inherited from class `hst_point` (by using some suitable super call). Any change on the definition of the same method in class `bd_point` will then be ignored by `bd_hst_point`. In a large program, such problems can easily pass unnoticed because the symmetry suggested by the multiple inheritance design would not be reflected by the actual implementation. This problem could be avoided by explicitly adding the class point as a base class for `bd_hst_point`. For example, in Eiffel we would need to rename (and discard!) the conflicting inherited methods of both base classes:

```
class
    bd_hst_point

inherit
    bd_point
        rename
            move as bp_move,
            print as bp_print
        end
    hst_point
        rename
            move as hp_move,
            print as hp_print
        end
    point
        redefine move, print end

feature
    print is
    do
        precursor
        print_bds
        print_hst
    end
    ...
end
```

This solution could also be implemented in C++ using virtual base classes:

```
class bd_hst_point
: public virtual point, public bd_point, public hst_point {
    ...
    void print();
    ...
}

void bd_hst_point::print()
{
    point::print();
    bd_point::print();
    hst_point::print();
}
```

This way, the class point will provide the basic behavior for the `move/2` and `print/0` methods. These two methods are redefined in order to include the needed calls to the methods inherited from classes `bd_point` and `hst_point` that implement the bounded behavior and the history behavior.

In Logtalk, we can use categories to solve this problem in a clean and extensible way without using multi-inheritance. In order to do so, we will start by defining two new categories, `bd_coord` and `point_hst`. The `bd_coord` category will contain the methods associated with point coordinate bounds:

```
:- category(bd_coord).

    :- public(set_bds/3).       % store a coordinate bounds
    :- public(bds/3).           % test/get coordinate bounds
    :- public(check_bds/2).     % checks coordinate value
    :- public(print_bds/1).     % print a coordinate bounds

    :- private(bds_/3).         % coordinate bounds storage
    :- dynamic(bds_/3).

    set_bds(Coord, Min, Max) :-
        ::retractall(bds_(Coord, _, _)),
        ::assertz(bds_(Coord, Min, Max)).

    bds(Coord, Min, Max) :-
        ::bds_(Coord, Min, Max).

    check_bds(Coord, Value) :-
        ::bds_(Coord, Min, Max),
        Value >= Min, Value =< Max.

    print_bds(Coord) :-
        ::bds_(Coord, Min, Max),
        writeq(bds(Coord)), write(': '), write((Min, Max)), nl.

:- end_category.
```

The methods for storing previous point positions will be encapsulated in the `point_hst` category:

```
:- category(point_hst).

    :- public(add_to_hst/1).   % store a point position
    :- public(init_hst/1).     % initialize position history
    :- public(hst/1).          % get the point history
    :- public(print_hst/0).    % print the point history

    :- private(hst_/1).        % position history storage
    :- dynamic(hst_/1).

    add_to_hst(Pos) :-
        ::retract(hst_(Hst)),
        ::assertz(hst_([Pos| Hst])).

    init_hst(Hst) :-
        ::retractall(hst_(_)),
        ::assertz(hst_(Hst)).

    hst(Hst) :-
        ::hst_(Hst).

    print_hst :-
        ::hst_(Hst),
        write('history: '), write(Hst), nl.

:- end_category.
```

Each one of the `bd_point`, `hst_point` and `bd_hst_point` classes will import the related categories in order to provide the intended behavior:

```
:- object(bd_point,
    imports(bd_coord),
    instantiates(class),
    specializes(point)).

    move(X, Y) :-
        ::check_bds(x, X),
        ::check_bds(y, Y),
        ^^move(X, Y).

    print :-
        ::print_bds(x),
        ::print_bds(y),
        ^^print.

:- end_object.
```

The same for the `hst_point` class:

```
:- object(hst_point,
    imports(point_hst),
    instantiates(class),
    specializes(point)).

    move(X, Y) :-
        ::position(X0, Y0),
        ^^move(X, Y),
        ::add_to_hst((X0, Y0)).

    print :-
        ::print_hst,
        ^^print.

:- end_object.
```

The `bd_hst_point` class will be defined as a point subclass, importing both `point_hst` and `bd_coord` categories:

```
:- object(bd_hst_point,
    imports(bd_coord, point_hst),
    instantiates(class),
    specializes(point)).

    move(X, Y) :-
        ::check_bds(x, X),
        ::check_bds(y, Y),
        ::position(X0, Y0),
        ^^move(X, Y),
        ::add_to_hst((X0, Y0)).

    print :-
        ::print_bds(x),
        ::print_bds(y),
        ::print_hst,
        ^^print.

:- end_object.
```

Note that the redefinition of our classes using the newly defined categories is transparent to the classes clients and descendants. Also, note that `bd_hst_point` is independent of both `bd_point` and `hst_point`, and yet it shares their behavior via the common imported categories. This solution can be easily extended if we need to add other point flavors besides coordinate bounds or position history. Using categories, we just import into an object each category implementing a desired flavor, no matter how many flavors and flavor combinations we may have. This compares favorably with a multi-inheritance solution where each new flavor will need to be implemented as a new sub-class, possibly inheriting from several other flavor sub-classes, resulting in high levels

of coupling between objects. We have thus found, not only an alternative solution to the use of multi-inheritance, but also a better one: a way to freely combine multiple orthogonal implementations without applying multi-inheritance mechanisms.

To better illustrate this example, here are some possible messages sent using the Logtalk top-level interpreter:

```
| ?- point::new(P,[xy-(1, 3)]).

P = p1
yes

| ?- p1::(print,  move(7, 4), print).

p1 @ (1, 3)
p1 @ (7, 4)

yes
```

Similar messages but with bounds on coordinate values:

```
| ?- bd_point::new(P, [xy-(1, 3), bds(x)-(0, 13), bds(y)-(-7, 7)]).

P = bp2
yes

| ?- bp2::(print, move(7, 4), print).

bds(x): 0,13
bds(y): -7,7
bp2 @ (1, 3)

bds(x): 0,13
bds(y): -7,7
bp2 @ (7, 4)

yes
```

Same problem but storing the history of past point positions:

```
| ?- hst_point::new(P, [xy-(1, 3)]).

P = hp3
yes

| ?- hp3::(print, move(7, 4), print).

history: []
hp3 @ (1, 3)

history: [(1,3)]
```

```
hp3 @ (7, 4)

yes
```

Same problem but with bounds on coordinate values and storing past positions:

```
| ?- bd_hst_point::new(P, [xy-(1, 3), bds(x)-(0, 13), bds(y)-(-7, 7)]).

P = bhp4
yes

| ?- bhp4::(print, move(7, 4), print).

bds(x): 0,13
bds(y): -7,7
history: []
bhp4 @ (1, 3)

bds(x): 0,13
bds(y): -7,7
history: [(1,3)]
bhp4 @ (7, 4)

yes
```

5.7 Summary

Logtalk categories are a simple and natural evolution of the original Smalltalk category concept, easily implemented by using the same compilation techniques that we apply to objects. Despite its simplicity, categories enable good solutions for reusing sets of utility methods and for implementing multi-inheritance designs.

Categories provide a way to encapsulate a set of related predicate definitions that do not represent an 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. As such, categories provide a straightforward solution for component-based programming in Logtalk.

The category concept is being actively used in the development of the Logtalk standard library and is being evaluated by writing Logtalk applications. Logtalk, viewed as an object-oriented programming research language, is a natural environment for trying out new ideas. However, adding a new feature to an established language must always be carefully pondered. I believe that the category concept here presented brings several important advantages to object-oriented languages and, in particular, to single inheritance languages.

Summarizing the main results, categories can be used to:

- Provide alternative solutions to the use of multi-inheritance for single-inheritance languages. Categories enable elegant implementations that minimize object coupling even in the context of multi-inheritance languages.

- Complement object variable-based composition by providing a composition mechanism where composed methods are at the same level as the container object methods, with full run-time transparency. Category imported methods are called, redefined, and otherwise used like any container object method.

- Enable different, unrelated objects, to share and reuse methods without using inheritance. This way methods can be made available to only those objects that really need them, avoiding large root objects populated with code that most descendants will never use.

- Split complex objects into a set of more manageable components, each containing a functionally cohesive set of methods that can be independently developed, compiled, and reused. Besides the advantages of incremental compilation, categories also make it possible to update an object without accessing its full source code.

- Encapsulate code that does not fit the notion of, or does not make sense as, an object. Two good examples are the Smalltalk dependency mechanism and the Logtalk monitoring methods.

It is also important not to forget the benefits that are inherited from the original Smalltalk-80 concept of methods functional categories, regarding code documentation and organization. Tacked together, all these features promote cleaner and simpler object-oriented designs and help improve code reuse across object-oriented applications.

An interesting research path will be to implement categories in common languages like Java or Smalltalk, either by using a preprocessor approach or by modifying an existing compiler. A good candidate would be a language like Squeak, a Smalltalk system written in Smalltalk itself, making it easy to modify it to try out new features. We expect categories to be easy to implement in dynamically type checked languages and, with some more work, in statically type checked languages like C++, retaining all the features presented in this chapter.

Future work may also include extending the Logtalk concept of categories to include some of the features of Objective-C categories, in particular, the possibility of augmenting a class protocol without modifying its source code. Note that Logtalk categories already enable us to update an object interface by updating an imported category, but the imports clause cannot be added or changed at runtime. A possible solution may be to adopt a mechanism to establish an import link without requiring sources changes on either importing objects or imported categories. Another possible, more explicit and declarative solution, would be to allow an object to import a set of matching categories. Extending the identity concept to allow category names with the same name but different arguments, we could then write:

```
:- object(an_obj,
    imports(a_ctg(_)).   % import all matching categories

    ...

:- end_object.
```

If we then have the following two matching categories:

```
:- category(a_ctg(foo)).

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

:- end_category.


:- category(a_ctg(bar)).

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

:- end_category.
```

Our object could then answer messages defined in all matching categories. For example:

```
| ?- an_obj::(foo(X), bar(Y)).
```

This solution is not difficult to implement but, as a language feature, it will probably be a source of misunderstandings, as it uses the same syntax of parametric objects with a very different semantics. It is also not clear that its benefits will outweigh the added complexity.

Chapter 6

Events

The concepts of events and monitors are an essential part of the inner workings of operating systems, graphical user interfaces, control and automation applications, and some programming languages and knowledge representation systems. Logtalk features events and monitors as language primitives. Therefore, Logtalk support for event-driven programming is at the same level as the support for object-oriented programming. Event-driven programming serves two main purposes. First, it complements the reflective capabilities of Logtalk. Second, it enables clean solutions for implementing complex object relations. Logtalk events are implemented at the message sending mechanism level. The implementation is fully portable across Prolog compilers and does not rely on the underlying operating system.

This chapter begins by presenting the Logtalk concepts of event and monitor, comparing them with related work, including access-oriented programming and Smalltalk dependency and event mechanisms. Secondly, the mechanisms for event generation and for communicating events to monitors are explained. Then, the Logtalk built-in predicates for event handling are described. Finally, some examples of event-driven programming are presented.

6.1 Events and monitors as language primitives

The words *event* and *monitor* have multiple meanings in computer science. Therefore, to avoid misunderstandings, this section starts by defining these concepts in the context of Logtalk event-driven programming.

6.1.1 Event definition

In an object-oriented application, all computations start by message sending. Consequently, sending a message can be interpreted as the primary event that occurs in these applications. Therefore, Logtalk defines an event as the sending of a message to an object.

By interpreting message processing as an atomic activity, the sending of a message, and the return of the control back to the *sender*, can be interpreted as two distinct events. This distinction is important as it enables a finer control over an application dynamics. In Logtalk, these two events are named *before* and *after* for message sending and return of control to the *sender*, respectively. An event can thus be represented by

the following ordered tuple:

```
(Event, Object, Message, Sender)
```

When one of the elements of this tuple is a free variable, or a term containing free variables, the tuple will specify not an event but a set of matching events. For example, the following tuple:

```
(after, list, _, _)
```

specifies all *after* events of any message sent by any object to the object `list`. Although processing a message results in two events being generated, the user defines which events are relevant and which objects should be notified of their occurrence. This translates into the concepts of event registration and monitor, which will be explained next.

6.1.2 Monitor definition

The concept of monitor complements the concept of event. In Logtalk, a monitor is an object that is automatically notified when a watched event occurs. The event notifications are performed by the runtime message sending mechanism and take the form of a call to the monitor event handlers. An event handler is a user-defined method whose arguments are the event description[1]. Event handling is performed by the method `before/3` for *before* events, and by the method `after/3` for *after* events. The *before* and *after* events associated to a message sending are independent. Therefore, a monitor may chose to be notified of the occurrence of either one or both events.

Conceptually, an event notification is performed by sending the corresponding message to the monitor object. Thus, event handling corresponds to the execution of a method — the event handler — in order to answer a message. In practice, the event handler call is determined and cached in the event registry table (discussed below) when the event is defined[2]. This optimization avoids constructing the event handler call at runtime, greatly improving performance (this and other implementation-related questions will be further discussed in Chapter 8).

Together with the definition of event, these definitions of monitor, event handler, and event notification, allow us to interpret the key concepts of event-driven programming in terms of object-oriented programming concepts. An alternative way of describing Logtalk event-driven programming is to see the runtime engine as a kind of meta-object defining a method implementing message processing. This method, coupled with an event registration table, is also responsible for sending the messages `before/3` and `after/3` to objects playing the role of monitors.

6.1.3 Event registration

For an object to act as a monitor, the events for which it needs to be notified must be registered with the runtime engine. The registration is accomplished by calling a Logtalk built-in predicate that will be described later on this chapter. Event registrations can either be performed by the monitor itself or by another object on its behalf. An event

[1] Event handlers are also known as *callbacks* on some programming languages.

[2] The compilation of the event handler call term can be regarded as an instance of static binding.

registration can be represented by the following ordered tuple:

```
(Event, Object, Message, Sender, Monitor, Handler)
```

The last tuple element, `Handler`, contains the compiled call to the monitor event handler.

The scope of the event registry table is global. This table is consulted by the message sending mechanism every time a message is processed in order to know which events to dispatch to which objects. Thus, the task of generating events and dispatching them to the registered monitors is performed by the message sending mechanism. Several monitors may need to be notified of the occurrence of the same event, each one for its own reasons. Therefore, the number of registered events is only bounded by the available computing resources.

An object acts as a monitor while there is an event table entry for it. Therefore, the monitor status of an object is toggled by registering and unregistering events. In addition, any static object can act as a monitor because the monitor status is not stored in the object itself.

6.1.4 Event-driven programming

The integration of object-oriented and event-driven programming aims to achieve the following goals:

- Provide a framework for building reflective applications based on the dynamic behavior of objects.

Event-driven programming provides support for *behavioral reflection*. Behavioral reflection can be defined as reflection about the dynamic behavior of an application resulting from message sending. Therefore, events complement the Logtalk support for *structural reflection*, which is provided by the built-in predicates and methods that allows us to perform reflective computations about entities, entity properties, entity predicates, and entity relations. Structural reflection can be defined as reflection about the structure of entities (meaning, its predicates) and about the relations between entities[3].

- Maximize object cohesion and minimize object coupling. An object should only contain its intrinsic properties. Objects should depend only on the public protocol of other objects, not on the protocol implementation.

Logtalk event-driven programming enables clean solutions for implementing dependency relations between objects. By defining dependent objects as monitors, their existence can be made transparent to the object they depend on. Furthermore, the methods of monitored objects do not need to contain any code anticipating possible dependency relations. These solutions provide several advantages over those found on other programming languages, as will be discussed next.

6.1.5 Related work

In this subsection a comparison is made between Logtalk event-driven programming and related work in other programming languages. Of particular relevance are the Smalltalk

[3]Note that these definitions of behavioral and structural reflection are a Logtalk interpretation of more general reflection concepts described, for example, in [95].

dependency and event mechanisms and their abstraction, the *observer* design pattern. Although there are other languages supporting some sort of event-driven programming, these Smalltalk mechanisms are the most influential ones in the design of the Logtalk event model. Comparisons with access-oriented programming and with CLOS standard method combination are also made.

Smalltalk dependency mechanism

Smalltalk-80 [19] and most of its descendant systems implement an event notification mechanism based on the notion of dependent object. A dependent object is an object that is notified when another object changes. The Smalltalk root class, `Object`, implements a set of methods to handle dependent objects. The method `addDependent:` adds an object to the list of dependent objects. The method `removeDependent:` removes an object from the list of dependent objects. The list of dependent objects can be retrieved (as a collection) using the method `dependents`. Two methods, `changed` and `changed:`, allow an object to inform all dependent objects of its changes. The implementation of these methods consists of sending the message `update:` to all dependent objects. The messages `changed` and `changed:` are sent to *self* by the instance methods that result in state changes. All dependent objects are notified of all changes, irrespective of the relevance a particular change might have to each of them. For applications in which dependent objects must be notified of any change on the observed object, this mechanism implements the desired behavior. For other applications in which objects depend only on specific aspects of the observed object, this mechanism is highly inefficient as it implies that each dependent object must parse every update message to verify its relevance.

Smalltalk event mechanism

In addition to the dependency mechanism presented above, most Smalltalk implementations include an event mechanism that allows dependent objects to register for specific events. In this mechanism, dependent objects are not added to the dependents list of the observed object. Instead, dependent objects register the events they are interested in with objects that are capable of triggering such events. When registering an event, a dependent object specifies the event triggering conditions, the message that should be sent, and the object that should be notified when the event occurs. To register an event, a dependent object sends the message `when:send:to:` to the object that it depends on. To trigger an event, an object sends to itself the message `triggerEvent:` (or one of its variants). This message plays a similar role to the message `changed:` in the dependency mechanism.

Logtalk events versus Smalltalk dependency and event mechanisms

Although both Smalltalk and Logtalk recognize the need for an event notification mechanism, the corresponding implementations differ in some important ways. While Logtalk events are implemented at the message sending mechanism level, Smalltalk dependency and event mechanisms are implemented at the class level. As such, Logtalk uses a global event registry table while Smalltalk stores dependent objects and events in the classes themselves.

In Smalltalk, the responsibility for notifying dependent objects belongs to the observed object methods. Therefore, an object has full control of when and how its dependent objects are notified. In addition, an object is responsible for defining which events it can trigger, and thus which events can be registered by dependent objects. This implies a level of object coupling between an observed object and its dependents, which does not exist in a Logtalk event-driven solution. In the development of Smalltalk applications whose objects will use the dependency or event mechanisms, we must add the calls that will trigger the event notifications to all relevant methods. However, the reuse of an existing class may require similar changes to its methods. This is possible only if the class source code is available and changes are allowed. An alternative will be to create a subclass where the relevant methods will be specialized in order to trigger the event notifications. None of these problems exists in Logtalk, where the existence of monitored events is completely transparent to the observed objects: Logtalk events are defined by the dependent objects. Moreover, the runtime engine is the sole responsible for event notification. Note that, in Logtalk, every public message is a potential event, while in Smalltalk (and in other programming languages that implement similar mechanisms) events are defined either implicitly (in the dependency mechanism) or explicitly (in the event mechanism) by the observed object.

While adding dependents to an object or registering events is easy, undoing these operations can be tricky and expensive. Depending on the Smalltalk implementation, garbage-collected dependent objects may or may not be automatically removed from the observer objects. Events may also need to be unregistered because a dependent object no longer needs to be notified of their occurrence. In this case, a message must be sent to all triggering objects to remove the dependent object. Unregistering events is straightforward in Logtalk due to its global event registry table. In addition, we can take advantage of pattern matching to delete all events related to a dependent object with a single call.

When evaluating performance, it should be noted that events add an overhead to every message sent using the `::/2` control construct. Although the current implementation is very efficient, this overhead exists even when no events are defined. The Smalltalk implementation of the dependency and event mechanisms limits its performance costs to the objects participating in the dependency relation.

As expected, the Logtalk and Smalltalk event-notification mechanisms implement different sets of features and trade-offs. However, the Logtalk support for event-driven programming does not preclude the use of Smalltalk-like dependency and event mechanisms. The Logtalk companion library already contains an implementation of the Smalltalk-80 dependency mechanism implemented through two categories that can be imported by any object.

Observer design pattern

The dependency and event mechanisms of Smalltalk can been abstracted as a reusable design pattern, usually named "Observer" [96], that is described as follows:

> "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically".

Besides Smalltalk implementations, variations of this design pattern can be also found on other object-oriented programming languages. For example, the Java API [97] provides

an implementation for this design pattern through the interface `java.util.Observer` and the class `java.util.Observable`. Most implementations of this design pattern, including the Java one, suffer from the problems already described for the Smalltalk dependency and event mechanism implementations.

Access-oriented programming

Event-driven programming and access-oriented programming share the concept of running a procedure whenever certain events occur. In access-oriented programming, procedures can be run whenever a variable is accessed or updated. These procedures are often named *daemons* on Artificial Intelligence systems such as KEE [17]. An example of a language supporting access-oriented programming is the Lisp multi-paradigm extension LOOPS [18]. Support for access-oriented programming can also be found on some Prolog object-oriented extensions such as SICStus Objects [24] and LPA Prolog++ [48, 47]. On SICStus Objects, *assert* and *retract* methods can be redefined to perform additional operations when an object database is updated. Prolog++ specifies two user-defined methods that are automatically called when an object attribute value is updated. The first method, named `invalid/2`, is used for validating the new attribute value. The second method, named `when_assigned/2`, is called upon successful assignment of the new attribute value.

Despite sharing the concepts of event and event-handler, there are some key differences between event-driven programming (as implemented in Logtalk) and access-oriented programming as found on SICStus Objects or Prolog++. These differences reflect the different scope of each programming technique. First, access-oriented programming events are related to variable access and update[4], while Logtalk interprets any message sent to an object as an event. Second, in Logtalk, any object can play the role of a monitor while in SICStus Objects and Prolog++ the event handlers are contained in the same object that contains the variable being accessed.

CLOS *before* and *after* methods

The Common Lisp Object System (CLOS) [54] defines *before* and *after* methods in the context of standard method combination. As such, these methods are not related to event-driven programming. They are mentioned here to avoid any misunderstandings with the use of the same names in Logtalk. Nevertheless, it is interesting to note the similarity between the Logtalk calling mechanism for event handlers and the CLOS calling mechanism for *before* and *after* methods. In both languages, sending a message implies calling all the corresponding *before* methods, followed by execution of the method selected to answer the message, and then all the corresponding *after* methods. A key difference is that, while in Logtalk any object can play the role of monitor and thus define *before* and *after* event handlers, CLOS *before* and *after* methods are attached to the classes along the inheritance chain containing the class defining the corresponding primary method. In addition, while in Logtalk the order of calling the event handlers is assumed arbitrary, CLOS *before* methods are called starting with the most specific one, whereas *after* methods are called starting with the most general one. Logtalk also provides built-in predicates for dynamically defining and abolishing events, thus toggling

[4] Prolog++ also supports attaching methods to instance creation and abolishing.

the role of monitor of an object and the call of its event handlers. There is no equivalent predefined functionality in CLOS.

6.2 Message sending and event generation

For each message sent using the `::/2` message sending control construct, the runtime engine automatically generates a *before* event and an *after* event. The *before* event is generated after validating the message but before calling the method selected to respond to it. In case the message is not valid, no *before* event is generated. The *after* event is generated after the selected method has successfully been executed. In case the selected method fails, the *after* event is not generated.

Messages sent using the message to *self* (`::/1`) or the *super* call (`^^/1`) control constructs do not generate events. The rationale behind this design choice is that messages to *self* and *super* calls are only used indirectly in the definition of methods or to execute additional messages within the same target object (represented by *self*). Therefore, events are only generated when an object uses the public protocol of another object. As such, an object cannot use events to break the encapsulation of other objects (for example, to find which methods are specialized by monitoring *super* calls or how methods are implemented by monitoring messages to *self*). Nevertheless, the workaround for generating events for messages sent to *self* is very simple:

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

In addition, if we need the sender of the message to be other than the object containing the predicate definition, we can write instead:

```
Predicate :-
    ...,
    self(Self),
    {Self::Message},
    ... .
```

The use of the `{}/1` control construct for encapsulating the message sending will result in the message being sent by the pseudo-object `user`.

Events are generated only for messages that correspond to user-defined methods. Calls to built-in methods do not generate events. This design choice was made because built-in methods are generally used only in the body of predicate definitions as messages to *self* (note that built-in methods cannot be redefined by the user). In addition, this allows us to optimize calls to built-in methods.

6.3 Communicating events to monitors

Whenever a watched event occurs, the message sending mechanism automatically notifies all monitors registered for that event. An event notification is performed by a direct

call to the corresponding monitor event handler. The event handler calls are made bypassing the message sending mechanism in order to improve performance and to avoid potential endless loops.

6.3.1 Defining event handlers

The event handler for *before* events is the predicate `before/3`. The event handler for *after* events is the predicate `after/3`. Any object acting as a monitor must define one or both of these event handler predicates, depending on the registered events:

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

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

The arguments of both methods are instantiated by the message sending mechanism when a watched event occurs.

The event handler methods have no declared scope. Therefore, they work as local, and thus private, predicates. In order to give an explicit scope to these methods, one may define a protocol containing the intended declarations. For example, the Logtalk library contains the following protocol:

```
:- protocol(event_handlersp).

    :- info([
        version is 1.0,
        author is 'Paulo Moura',
        date is 2000/7/24,
        comment is 'Event handlers protocol.']).

    :- public(before/3).
    :- mode(before(@object, @nonvar, @object), zero_or_one).
    :- info(before/3, [
        comment is 'Event handler for before events.',
        argnames is ['Object', 'Message', 'Sender']]).

    :- public(after/3).
    :- mode(after(@object, @nonvar, @object), zero_or_one).
    :- info(after/3, [
        comment is 'Event handler for after events.',
        argnames is ['Object', 'Message', 'Sender']]).

:- end_protocol.
```

Using this protocol, our monitor objects would be defined as follows:

```
:- object(monitor,
    implements(event_handlersp), ...).
    ...
:- end_object.
```

Declaring event handlers as public or protected predicates enables the specialization of event handler definitions inherited from an ancestor object[5]. However, declaring the event handlers as public predicates is not generally recommended, as these predicates should only be called through the message sending mechanism. The alternative will be to write:

```
:- object(monitor,
    implements(protected::event_handlersp),
    ...).
    ...
:- end_object.
```

This will make the scope of the monitor event handlers protected instead of public.

6.3.2 Event handler semantics

The established semantics for event handlers is that they must succeed as a necessary condition so that the corresponding message can succeed. Specifically:

- All event handlers associated to a *before* event must succeed so that the message can be sent. The failure of a *before* event handler prevents the execution of the remaining event handlers and further message processing.

- All event handlers associated to an *after* event must succeed so that the message itself succeeds. The failure of any event handler associated to an *after* event forces backtracking over the message execution.

- The failure of an event handler never results on backtracking over the execution of the preceding event handlers.

- The arguments of the event handlers are strictly input arguments. Any further instantiation of these arguments resulting from the execution of an event handler is discarded before proceeding to the next event handler or processing the watched message.

These are the most generic design choices under the constraints of the event and monitor properties described earlier. The independence between monitors implies that the failure of an event handler must not result on backtracking over the event handlers executed before it. In addition, as multiple monitors may be notified of the same event, further instantiation of event handler arguments must be discarded to ensure that the results of the processing of a message does not depend on the order of event handler calls.

The semantics of *before* event handlers allows them to be used as guards for message execution. The semantics of *after* event handlers allows them to be used for testing, and possibly rejecting, message results. The existence of a monitor can be made transparent to message processing by defining its event handlers so that they never fail (which is trivial to accomplish).

[5]The *super* call, `^^/1`, always checks the predicate declaration of its argument.

6.3.3 Activation order of event handlers

Ideally, whenever there are several monitors defined for the same event, the calling order of the respective event handlers should be irrelevant for the final result. However, this is not always possible. If an event handler has side effects (for example, input/output operations or object state changes), different calling orders may lead to different results when an event handler fails, as the following event handlers will no longer be activated. Moreover, the order of event handler activation should be assumed as being arbitrary. To assume or to try to impose a specific sequence implies a global knowledge of an application dynamics, which is not always possible. In addition, that knowledge can reveal itself as incorrect if the execution conditions change.

6.4 Event registration

Logtalk provides three built-in predicates for handling the event registration table. These predicates enable us to find which events are defined, to define new events, and to abolish events when they are no longer necessary. For applications that rely heavily on event-driven programming, the best approach will be to define a set of objects that will use the built-in predicates described below to implement higher-level behavior.

6.4.1 Defining new events

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

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

Only the last argument, `Monitor`, needs to be instantiated when calling this predicate. For example, the following call:

```
| ?- define_events(after, _, _, _, monitor).
```

sets the object `monitor` to act as a monitor for the *after* events of every message sending. Thus, when called with arguments containing free variables, this predicate defines not an event but a set of matching events.

6.4.2 Abolishing defined events

Events that should no longer be spied may be abolished by calling the built-in predicate `abolish_events/5`:

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

Calling the predicate with free variables will remove all matching events.

6.4.3 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 may return not a single event but a set of matching events if free variables are used in the definition of new events.

6.5 Examples

This section presents three simple examples of event-driven programming in Logtalk. The first two examples illustrate the implementation of common reflective applications, such as debuggers and profilers, using events and monitors. The third example shows how relations between objects can be implemented using events to maintain relation constraints. The full source code of these (and similar) examples is available in the current Logtalk distribution.

6.5.1 Tracing messages

Assume that we want to track any message sent to an object by printing a descriptive text to the standard output. This tracing behavior can be easily implemented by making every message sending an event, and by defining a monitor that will print the required text. A possible definition would be:

```
:- object(tracer).

    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.
```

After compiling and loading this object, 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. To illustrate this tracing behavior, we can use the objects from the metapredicates example in Chapter 3, page 60. After compiling and loading the objects `list` and `sort(_)` (first removing the calls to the metapredicate `trace/1`), using the same query of the original example will result in the following output:

```
| ?- sort(user)::sort([3, 1, 4, 2, 9], Sorted).

call: sort(user) <- sort([3,1,4,2,9],_18) from user
call: list <- append([],[2],_132) from sort(user)
exit: list <- append([],[2],[2]) from sort(user)
call: list <- append([],[1,2],_157) from sort(user)
exit: list <- append([],[1,2],[1,2]) from sort(user)
call: list <- append([],[9],_200) from sort(user)
exit: list <- append([],[9],[9]) from sort(user)
```

```
call: list <- append([],[4,9],_225) from sort(user)
exit: list <- append([],[4,9],[4,9]) from sort(user)
call: list <- append([1,2],[3,4,9],_18) from sort(user)
exit: list <- append([1,2],[3,4,9],[1,2,3,4,9]) from sort(user)
exit: sort(user) <- sort([3,1,4,2,9],[1,2,3,4,9]) from user

Sorted = [1,2,3,4,9]
yes
```

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

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

yes
```

This is a simple example of how events and monitors can be used for debugging Logtalk programs. The current Logtalk library contains objects implementing a full debugger, modeled after the four ports box control flow model common of Prolog debuggers, which allows the programmers to trace and spy objects and messages.

6.5.2 Profiling

We can use events for profiling our programs. Assume, for example, that we want to measure message execution time. We can define a new object, named `stopwatch`, containing definitions for the `before/3` and `after/3` event handlers that will print, respectively, the starting system time and ending system time whenever a spied message is sent:

```
:- object(stopwatch).

    before(Object, Message, Sender) :-
        time::cpu_time(Seconds),
        write(Object), write(' <-- '), writeq(Message),
        write(' from '), write(Sender), nl,
        write('start: '),
        write(Seconds), write(' seconds'), nl.

    after(Object, Message, Sender) :-
        time::cpu_time(Seconds),
        write(Object), write(' <-- '), writeq(Message),
        write(' from '), write(Sender), nl,
        write('end:   '),
        write(Seconds), write(' seconds'), nl.

:- end_object.
```

After compiling and loading the object `stopwatch`, we can profile, for example, every message from any object to an object named `list` by making the appropriate call to the built-in predicate `define_events/5`:

```
| ?- define_events(_, list, _, _, stopwatch).
```

From now on, we will get the starting and ending execution time whenever a message is sent to the object `list`. For example:

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

list <-- length([0,1,2,3,4,5,6,7,8,9], _7) from user
start: 1234.79946798 seconds
list <-- length([0,1,2,3,4,5,6,7,8,9], 10) from user
end:   1234.80048351 seconds

Length = 10
yes
```

When no more timing information is needed, we can remove the event definitions by calling the built-in predicate `abolish_events/5`:

```
| ?- abolish_events(_, list, _, _, stopwatch).
```

This simple example can be easily extended. Most Prolog systems give access to several data regarding memory usage and timing information that can be used to define powerful profilers for our programs.

6.5.3 Constrained object relations: a stack of blocks

Object relations may imply constraints on the state and behavior of participating objects. The representation of these relations must minimize object coupling between the objects representing the relation and the objects participating on the relation. Logtalk support for event-driven programming allows us to represent these dependency relations between objects without breaking object encapsulation and minimizing object coupling.

Assume that we want to represent stacks of blocks, where each stack will comply with the following two rules:

1. When moving a block, all blocks on top of it will move accordingly.
2. When moving a block that is on top of other block, the stacking relation between them will be broken and the corresponding relation tuple will be deleted.

In order to simplify this example, it is assumed that all the blocks have the same size and that they exist in a two-dimensional world. In addition, assuming that only one block can be stacked on the top of another block, we can represent a stack as a binary relation between blocks.

This example adopts a class-based solution. Blocks will be represented as instances of class `block`, while the stacks of blocks will be represented by the instance `block_stack`. Assuming that the base classes presented in Chapter 1 for building reflective class-based applications, the class `block` could be defined as follows:

```
:- object(block,
    instantiates(class),
    specializes(object)).

    :- public(position/2).
```

```
    :- public(move/2).

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

    position(X, Y) :-
        ::position_(X, Y).

    move(X, Y) :-
        ::retractall(position_(_, _)),
        ::assertz(position_(X, Y)).

:- end_object.
```

Note that all the predicates contained in this object are intrinsic to it. No assumptions are made about possible relations that the `block` instances may participate into. Specifically, the definition of the method `move/2` does not contain any call to notify other objects of state changes, as it would be necessary in a solution based on the Smalltalk dependency or event mechanisms.

Assume that an object named `relation`, providing basic functionality to represent relations between objects, is already defined as follows:

```
:- object(relation,
    instantiates(class),
    specializes(object)).

    :- public(tuple/1).          % table of relation tuples
    :- public(add_tuple/1).      % add a new relation tuple
    :- public(remove_tuple/1).   % remove a relation tuple
    ...

:- end_object.
```

We can now define `block_stack` as an instance of class `relation`:

```
:- object(block_stack,
    instantiates(relation)).

    add_tuple([A, B]) :-         % add a new tuple by first moving
        B::position(Xb, Yb),     % the top block A to the top of
        Ya2 is Yb + 1,           % block B and set block_stack as
        {A::move(Xb, Ya2)},      % a monitor for both blocks
        ^^add_tuple([A, B]),
        define_events(after, A, move(_, _), _, block_stack),
        define_events(after, B, move(_, _), _, block_stack).

    ...

    after(A, move(X, Y), Sender) :-  % propagate changes to stack
        \+ this(Sender),             % blocks, avoiding loops:
```

```
        ::tuple([A, B]), !,          % remove tuple if block A no
        Y2 is Y - 1,                 % longer on top of block B
        (B::position(X, Y2) ->
            true
            ;
            ::remove_tuple([A, B])).

    after(B, move(X, Y), Sender) :-  % propagate changes to stack
     \+ this(Sender),                % blocks, avoiding loops:
        ::tuple([A, B]), !,          % move block on top of block B
        Y2 is Y + 1,                 % to its new position
        {A::move(X, Y2)}.

:- end_object.
```

Note that the object `block_stack` depends only on the public protocol of `block` instances (`position/2` and `move/2`). In this object, the event handler `after/3` plays a similar role to the Smalltalk `update:` method. The difference is that, while in Logtalk the method `after/3` is called by the runtime engine, in Smalltalk the method `update:` would be called (indirectly via sending to *self* the message `changed:`) by the methods of class `block`.

In order to make this example easier to follow, we can take advantage of Logtalk event-driven programming to define an object, `stack_monitor`, which will print an ASCII representation of all blocks to the standard output every time one of them is moved to a new position:

```
:- object(stack_monitor).

    after(_, move(_, _), _) :-    % pretty prints ascii
        ... .                     % stack representation

:- end_object.
```

After compiling and loading all the above objects, we can start by creating four `block` instances, each one in a different initial position:

```
| ?- block::new(a, [position-(8, 1)]).
yes

| ?- block::new(b, [position-(6, 1)]).
yes

| ?- block::new(c, [position-(4, 1)]).
yes

| ?- block::new(d, [position-(2, 1)]).
yes
```

Before beginning to construct a stack of blocks by using the message `add_tuple/1`, we can set the object `stack_monitor` as a monitor for all `move/2` messages sent by any object through the following call:

```
| ?- define_events(after, _, move(_,_), _, stack_monitor).
yes
```

To stack the four blocks we have created, we will need the following three calls:

```
| ?- block_stack::add_tuple([c,d]).
|.c......
|.d...b.a
---------
yes

| ?- block_stack::add_tuple([b,c]).
|.b......
|.c......
|.d.....a
---------
yes

| ?- block_stack::add_tuple([a,b]).
|.a
|.b
|.c
|.d
---
yes
```

The above calls set `block_stack` as a monitor for our four blocks:

```
| ?- current_events(after, Block, _, _, block_stack).
Block = a ;
Block = b ;
Block = c ;
Block = d
yes
```

As a first test, we can move the whole stack to a new position by moving its bottom block:

```
| ?- d::move(9, 1).
|.a.......
|.b.......
|.c.......
|........d
----------
|.a.......
|.b.......
|........c
|........d
----------
|.a.......
```

```
|........b
|........c
|........d
----------
|........a
|........b
|........c
|........d
----------
yes
```

Note that `stack_monitor` will print the current position of all blocks every time one of them is moved. As expected, moving the block `d` triggers `block_stack` to move the block on top of it, `c`, which in turn triggers the block `b` to move, which triggers the move of the block `a`.

As a second test, we can break the stack in half by moving the middle block `b` to the "ground":

```
| ?- b::move(3, 1).
|........a
|.........
|........c
|..b.....d
----------
|..a.....c
|..b.....d
----------
yes
```

As in the first test, the move of the block `b` triggers `block_stack` to move accordingly the block `a`. In addition, the tuple `[b, c]` will be removed as it will no longer be valid, as we can check with the following goal:

```
| ?- block_stack::tuple(Tuple), write(Tuple), nl, fail.
[c,d]
[a,b]
no
```

The stack can be folded back by creating, for example, the tuple `[d, a]`:

```
| ?- block_stack::add_tuple([d, a]).
|..d......
|..a.....c
|..b......
----------
|..c
|..d
|..a
|..b
----
yes
```

As a last test, we can once again move the whole stack to a new position by moving its bottom block:

```
| ?- b::move(5, 1).
|..c..
|..d..
|..a..
|....b
------
|..c..
|..d..
|....a
|....b
------
|..c..
|....d
|....a
|....b
------
|....c
|....d
|....a
|....b
------
yes
```

This example could be easily extended to deal with other constraints on the stacked blocks. For example, `block_stack` could be set to also monitor the deletion of blocks and to act accordingly. The Logtalk library contains classes for representation of relations and constrained relations between objects. The example presented above and similar examples (e.g. concentric relations between geometric shapes) can be found on the current Logtalk distribution.

6.6 Summary

Logtalk support for event-driven programming has been achieved by reinterpreting the concepts of event, monitor, event notification, and event handling in the context of object-oriented programming. This reinterpretation allows us to view event-driven programming as an instance of object-oriented programming, providing a conceptual integration of both programming paradigms. This translates into the implementation of the event concepts as primitive language features, at the same level as objects and messages. Logtalk therefore contrasts with other programming languages, such as Smalltalk and Java, where those concepts are implemented at the application level. The implementation of events as a language intrinsic feature rises some performance issues. These will be further discussed in Chapter 8.

Through event-driven programming, objects gain autonomy to act, i.e. they are no longer restricted to being activated only when they receive a message. Events can be used to log, react, control, and modify an application behavior.

Events complete the Logtalk framework for reflective computing. Specifically, events provide Logtalk with support for behavioral reflection, complementing the support for structural reflection provided by the Logtalk built-in predicates and built-in methods already described in the previous chapters. Therefore, reflective applications, such as debuggers and profilers, are easily defined in Logtalk.

Events allow the implementation of dependency relations between objects without breaking object encapsulation. By defining dependent objects as monitors, the methods of observed objects do not need contain any calls to event-related methods or any other code that is not intrinsic to their nature. This maximizes object cohesion and, at the same time, minimizes object coupling because dependent objects rely only on the public protocol of observed objects. Therefore, any object can be (re)used — without any modification — in applications where it will play the role of an observed object. This is an important advantage of Logtalk event-driven programming solutions over Smalltalk-based dependency and event mechanisms, which imply that the code of observed objects take into account their participation in dependency relations. Logtalk event-driven programming provides a solution for reusing objects whose methods do not support a Smalltalk-type solution, and cannot be modified to add that support. Nevertheless, Smalltalk-like dependency and event mechanisms are easily implemented in Logtalk through library protocols and categories. Coupled with Logtalk event-driven support, the user can choose the best solution for representing dependencies between objects on a case-by-case basis.

Chapter 7

Documenting Logtalk programs

Logtalk provides support for automatic program documentation. By default, compiling an object, category, or protocol, generates a documenting file containing essential information about the compiled entity. It is easier to maintain source code and its documentation synchronized, if they are in the same file instead of in separated files. Logtalk documenting features are focused on versioning and interface description information and not on documenting algorithms and other high level code descriptions common to literate programming tools [98].

This chapter begins by explaining the documenting language design choices. Secondly, the documenting file format is explained. Next, the Logtalk language support for expressing arbitrary documenting information is presented. Finally, the automatic generation of documenting files in the current Logtalk implementation is described.

7.1 Documenting language

Automatic generation of documenting files implies support both at the language level and at the compiler level. At the language level, we must be able to represent arbitrary information about an entity and its predicates. At the compiler level, all relevant information must be parsed, formatted, and written out to a documentation file.

Two common examples of automatic documenting tools are `Javadoc` [99] and `Lpdoc` [100]. The first is a standard tool included in the Java SDK. The second is a tool for documenting (C)LP systems, including Prolog programs. `Javadoc` uses specially formatted program comments to allow the user to specify documenting information. This means that two languages will be used when writing programs: one language for code and another language for documentation. This is an approach shared by most literate programming tools such as `CWEB` [101] and its offsprings. `Lpdoc` defines a sophisticated *assertion language* that can be seen as an extension of Prolog, made of a set of directives. In both cases, we end up using two different tools: a language compiler and a separate tool for extracting program documentation.

In Logtalk, an important design decision was to use the same language for both code and documentation. After all, Logtalk is a declarative language. All documenting information in Logtalk source files is expressed using Logtalk directives. That is, documenting directives are an integral part of the Logtalk language. This implies that the Logtalk compiler is responsible for parsing and extracting all relevant documenting information when an entity is compiled. Expressing documenting information in the

Logtalk language itself, instead of extending it with foreign directives or using specially formatted comments, has several advantages. First, any Logtalk code with relevant documentation meaning can be easily summarized without being first rewritten in some documenting language. Second, representing documentation by code (arguably much easier and more feasible in declarative languages such as Prolog and Logtalk than in most other programming languages), means that any integrated development tool must understand only one language to access all the content of our programs. Third, it makes program documentation a question of language design, instead of an afterthought disconnected from the language specification. An immediate consequence of this design choice is that the compiler is responsible for the generation of documentation files. This raises two questions. First, we must choose a documenting file format simple to generate and easy to convert to any desirable human or machine-readable format. Second, we must design the Logtalk documenting directives so that the programmer can represent whatever information is needed without the need of revising the Logtalk language specification. I will address these two questions in the following sections.

7.2 Documenting file format

The first generation of Logtalk (`1.x`) [27], released in 1995, included a set of objects that generated object documentation in plain text, HTML [102], and LaTeX2e [103] formats. This set of objects shared the same basic structure but the output formats were different enough to prevent significant code sharing. Generating documentation in another format often implied writing a new object, even if we only needed a variation of the HTML or LaTeX2e output formats. Since then, the development of the XML standard [104] provided a much better and flexible choice for the output of documentation files. XML can be seen as a meta-language, allowing the definition of our own markup languages, ideally suited for the representation of structured information. A common use is the interchange of information between databases and legacy systems. Instead of generating documentation in a final format (for printing or for on-line viewing), the current Logtalk generation outputs a XML file when compiling an entity containing all the relevant documenting information. 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 (including name, compilation mode, scope, and call modes). The documentation files can be enriched with arbitrary user-defined information, either about an entity or about its predicates, by using two sets of directives described in the next section. The structure of this documenting file is defined in the Logtalk compiler. A formal specification of the documenting file format, using both Document Type Definition (DTD) [105] and XML Schema [106] syntaxes, is described in the appendix C. Representing entity documentation in XML format allows us to use the growing set of XSL [107] tools to process and convert the documenting files to any desirable format. For example, the current Logtalk release includes all the necessary files to transform documenting XML files into HTML, LaTeX2e, and PDF [108] files. The appendix C contains some XSL transformation files examples.

7.3 Documenting directives

Logtalk defines two documenting-specific directives for providing arbitrary user-defined information about an entity and about its predicates. These two directives complement other Logtalk directives that also provide important documentation information, such as predicate scope and predicate call modes.

7.3.1 Entity documenting directives

As implied by its name, entity documenting directives are used to represent information regarding an entity as a whole. Let us first note that, in the context of documenting an entity, every entity directive provides relevant information. For example, the entity opening directive describes the entity type (object, category, or protocol) and the relationships between the entity and other entities. Logtalk defines also an entity documenting-specific directive, described next, enabling the representation of common information such as entity creation and modification dates, entity author, or a short entity textual description.

The `info/1` directive

Arbitrary user-defined entity information can be represented using the directive `info/1`:

```
:- info([
    Key1 is Value1,
    Key2 is Value2,
    ...]).
```

In this template, keys must be atoms and values must be ground terms. The following keys should be considered as predefined and may be processed specially by Logtalk:

`comment`
: Comment describing entity purpose (an atom).

`author`
: Entity author (an atom).

`version`
: Version number (a number, preferably an integer).

`date`
: Date of last modification (formatted as `Year/Month/Day`).

`parnames`
: Parameter names for parametric entities (a list of atoms).

Of course, we should only use the keywords that make sense for our application, remembering that we are free to invent our own keywords. 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']).
```

Here we have used a `diagram` keyword to refer to some hypothetical class diagram.

Object dependencies

In addiction the relations declared in a category or object-opening directive, the predicate definitions contained in the entity may also imply other dependencies. These can be documented by using the `calls/1` and the `uses/1` directives.

The `calls/1` directive can be used when a predicate definition sends a message that is declared in a specific protocol:

```
:- calls(Protocol).
```

If a predicate definition sends a message to a specific object, this dependence can be declared with the `uses/1` directive:

```
:- uses(Object).
```

These two directives are used by the Logtalk runtime engine to check if all needed entities are loaded when running an application. They may be also used in a future version to implement object namespaces.

7.3.2 Predicate documenting directives

Predicate documenting directives are used to represent information regarding a specific entity predicate. Essential predicate information is provided in the predicate scope and predicate mode directives. This information can be complemented using the directive described next.

The `info/2` directive

Arbitrary user-defined predicate information can be represented using the directive `info/2`:

```
:- info(Functor/Arity, [
    Key1 is Value1,
    Key2 is Value2,
    ...]).
```

In this template, the first argument identifies the predicate. The second argument is a list of key-value pairs where keys must be atoms and values must be ground terms. The following keys and key values are predefined and may be processed specially by Logtalk:

`comment`
: Comment describing predicate purpose (an atom).

`argnames`
: Names of predicate arguments for pretty print output (a list of atoms).

`allocation`
: Objects where we should define the predicate. Some of the possible values are `container`, `descendants`, `leafs`, `instances`, `classes`, `subclasses`, and `any`.

`redefinition`
: Describes if and how a predicate can be redefined. Some of the possible values are `never`, `free`, `specialize`, `call_super_first`, `call_super_last`.

The possible values for the `redefinition` and `allocation` keys deserve further comments.

Regarding the `redefinition` key, the value `never` means that the predicate should not be redefined. The opposite value, `free`, means that the inherited definition can be freely overridden. The remaining three values imply that the inherited definition must be specialized. The more general value, `specialize`, just specifies that the new definition must call the inherited definition (using a `^^/2` call). The last two values, `call_super_first` and `call_super_last`, specify that the inherited definition must be called, respectively, either as the first or as the last call in the new definition.

The `allocation` key describes where a declared predicate should be defined. The `container` value means that the predicate is defined in the same object that contains the predicate declaration. The `any` value is used when the predicate can be defined in the object containing the declaration or in any of its descendants. The `instances`, `classes`, and `subclasses` applies only to class-based hierarchies. The `classes` value is similar to the `any` value, except that the predicate should not be defined in instances, only in its declaring class and respective sub-classes. The `instances` and `classes` values have similar meanings. The `descendants` and `leafs` can be applied to both class-based and prototype-based hierarchies. The `descendants` value means that the definition can be contained in any descendant of the object declaring the predicate, while the `leafs` value implies that predicate should only be defined in the descendant leafs of the hierarchy rooted in the object declaring the predicate.

Note that this set of keys and values cover the most common scenarios. However, we can always define our own keywords and/or additional keyword values. For example:

```
:- info(color/1, [
    comment is 'Table of defined colors.',
    argnames is ['Color'],
    allocation is instances,
    constraint is 'At most four visible colors allowed.']).
```

Here we have defined a `constraint` key to express some predicate constraint.

7.4 Processing and viewing documenting files

The actual details of processing and viewing documenting files are necessarily out-of-scope of the definition of Logtalk as a programming language. Nevertheless, we describe here briefly the processing flow of documenting files in the current Logtalk implementation.

As stated before, the XML documenting files are (by default) automatically generated when we compile a Logtalk entity. For example, assuming the default filename extensions used by the Logtalk compiler, compiling a `list.lgt` source file generates a `list.pl` Prolog file and a `list.xml` XML file.

Every XML file contains references to two other files: `logtalk.dtd`, a DTD file describing the XML file structure, and a XSLT style sheet file responsible for converting the XML files to some desirable format such as HTML. The name of the XSLT file can be changed either system-wide in a configuration file, changed for a duration of a programming session by setting a compiler flag, or defined in per-file basis using a compiler option. The default value is `lgtxml.xsl`, a XSLT file for Web browser

conversion of Logtalk XML files to HTML output whose links point to related XML files. The HTML output refers a CSS file, `logtalk.css`, which specifies how the HTML code will be rendered. For on-line viewing of the XML documenting files, one can open them on a web browser that supports the XML, XSLT, CSS 1, and HTML 4 standards or use a XSLT tool to compile the `.xml` files to `.html` files. If we want a printed version of the documentation, we can use one of the supplied XSLT files to convert the XML code to either PDF or LaTeX2e files. One can also write new XSLT files to convert the XML code to other alternative formats. The appendix C provides further details and some examples.

Logtalk provides a set of compiler options to control the generation of entity documenting files. These options will be fully described in Chapter 8.

7.5 Summary

Tools such as `Lpdoc` and some Prolog compilers define documenting directives similar to the Logtalk directives `info/1` and `info/2` presented in this chapter. For example, `Lpdoc` and ECLiPSe [63] both define a directive named `comment/2` that subsumes both the entity and the predicate Logtalk documenting directives. As in Logtalk, these directives correspond to no-operational code that is discarded by the compiler after parsing. `Lpdoc`, ECLiPSe, and Logtalk share some of the keys used in the respective documenting directives, although some Logtalk keys are only meaningful in an object-oriented context. Documenting directives is a topic where some sort of standardization in the Prolog community would be useful, enabling the development of cross-compiler document extracting and formatting tools. Reaching an agreement should be relatively easy due to the no-operational feature of the documenting directives.

Regarding documenting file formats, an advantage of the Logtalk solution for the automatic generation of documenting files is the use of XML to express the documenting information. This allow us to easily convert the documenting files to any application or domain specific language, including common online and printing formats such as HTML, PDF, LaTeX2e, or to any other format that proves popular in the near future, such as the emerging XHTML standard [109]. XML is also an ideal format for information interchange between systems. Flexibility is the key concept here. The important point is that the XML files contain only the documenting information, without being cluttered with any kind of presenting or formatting information. By contrast, `Lpdoc` generates `Texinfo` documenting files, a format that is biased towards the generation of both online and printing formats from the same source file. In fact, a `Texinfo` file structure is similar to LaTeX files or traditional books, with chapters and sections. In ECLiPSe, a library provides the tools needed to generate HTML files from the Prolog source files. Although we can convert HTML to other formats, this is a final delivery format for online reading, not an information interchange format.

Another Logtalk advantage over other documenting tools is the design choice of expressing Logtalk source file documentation in Logtalk itself. There is no need of using one tool for extracting program documentation and another tool to compile the source file: there is only one language. Both code compilation and documentation extraction are performed by the Logtalk compiler. It can be argued that this makes the compiler more complex because it must perform two functions: code generation and documentation extraction. However, the compiler only needs to parse one language

for both documentation and code. This makes an easy task to extract documenting information from language constructs that are not documenting specific such as predicate declarations or inheritance relations. I should note that the documenting files may be further enriched by the programmer with more documenting information using any of the many available XML editor tools that automatically generates an entry form from a DTD or a XML Schema specification.

As a final remark, a possible long term solution for the documentation of both Logtalk and Prolog source files would be to adopt some of the extensive work accomplished on the `Lpdoc` tool by extending it to document object-oriented information, using XML as the documenting file format. However, such task is outside the scope of this thesis work.

Chapter 8

Implementation

This chapter provides a high-level description of the current Logtalk language implementation. The implementation consists of a compiler and a runtime engine. These two components are combined in a single program in order to support the dynamic features of Logtalk that allow us to dynamically create and modify objects, protocols, and categories at runtime.

The implementation solutions described in this chapter are based on the works of Francis McCabe [23] and Markus Fromherz [52], extended to support the richer set of object-oriented features of Logtalk.

This chapter begins by describing the design choices made for the current Logtalk implementation. Secondly, an overview of the implementation is presented. Thirdly, the compilation of entities, entity relations, entity directives, predicate directives, and predicate clauses is explained. Fourthly, the runtime engine support for events and monitors is examined. Next, some of the limitations of the current implementation are described. Finally, the issues found while porting Logtalk are discussed.

8.1 Design choices

Converting scientific and technical goals into an actual implementation implied a set of design choices that will be presented in this section. Different choices could have been made without affecting, in any fundamental way, the specification of the Logtalk language presented on this thesis. Nevertheless, these design choices have practical consequences for the language implementation. They determine how the language is made available, how it is used and for what kinds of applications, and how it integrates with existing Prolog compilers. These choices represent a set of trade-offs between performance and flexibility of development and between strengths and limitations.

8.1.1 Logtalk as a Prolog preprocessor

Logtalk, as an extension to Prolog, can be implemented in three different ways, in increasing degree of difficulty:

Modification of an existing Prolog compiler
: Most open source Prolog compilers could easily be extended with a Logtalk implementation. This would provide the best user experience due to the high level

of integration achieved. However, this would also restrict Logtalk compatibility to the modified Prolog compiler.

Preprocessor for existing Prolog compilers
Logtalk could also be implemented as a Prolog program that would act as a preprocessor when loaded. This option would allow users to continue to use their favorite Prolog compilers. The drawback of this approach would be the amount of porting work that would need to be performed per Prolog compiler.

Full, independent implementation
Implementing Logtalk from scratch would have the advantage of full control on the implementation, including its code distribution and use license. However, this option would also be the most difficult one, as it would imply the implementation of a full ISO standard compliant Prolog subsystem. Note that in the first two options, Logtalk can take advantage of the Prolog compiler native parser to read and parse source files.

One of the Logtalk technical goals is the compatibility with existing Prolog compilers, the ISO Prolog standard, and operating systems. As such, I chose the second design option, implementing Logtalk as a Prolog preprocessor. Thus, the current Logtalk implementation compiles Logtalk source files to Prolog source files, which in turn are further compiled by a Prolog compiler.

8.1.2 Compatibility and portability

Writing a portable program implies finding a common subset of features of the selected Prolog compilers. Usually, this means giving up things such as non-basic access to the operating system, graphical user interfaces, and interfaces with other languages. Moreover, sometimes we will find ourselves patching compilers, redefining problematic built-in predicates, and changing predefined operator priority and types. In doing so, our program may become incompatible with other programs written for the same compiler.

ISO Prolog standard as abstract target

The subset of features chosen is the one specified in the Part I of the ISO Prolog standard. The major reason for this choice was the quality of the standard documentation. It provides a detailed specification of Prolog behavior and Prolog built-in predicates that can be used to write a portable Logtalk implementation. Thus, conformance with the ISO standard will be the minimal requirement for Prolog compiler compatibility with Logtalk. Specifically, Logtalk must be able to run feature-complete in any standard-compliant Prolog compiler. However, conformance here should be understood in a loose sense, as only a few Prolog compilers comply with the ISO standard fully.

Logtalk configuration files

The interface between Logtalk and a specific compiler is accomplished through a *configuration file*. This is a plain Prolog file that must be loaded before the Logtalk system[1]. For standard compliant compilers, the configuration file is minimal containing only some glue code and utility predicates. For older or less compliant compilers, the configuration file also contains workaround definitions for missing ISO Prolog standard predicates.

[1]The Logtalk system includes both the preprocessor and the runtime engine.

8.1.3 Dynamic binding

Logtalk uses dynamic binding for message processing. This means that the lookup searches of both the predicate declaration necessary to validate a message and the predicate definition necessary to answer it are performed at runtime.

8.1.4 Static relations between entities

The entity relations declared in the entity-opening directives are static and cannot be changed at runtime. This is a common characteristic of most class-based object-oriented languages. However, some prototype-based languages allow the dynamic reclassification of objects. Although there is no native support in Logtalk for this feature, it is possible to implement it using dynamic entities and the built-in reflection methods. This design choice allows the representation of entity relations to be optimized for predicate lookup searches. This is critical for good system performance (due to the use of dynamic binding, predicate lookup searches are performed every time a message is sent).

8.1.5 Independent entity compilation

In the current implementation, all entities (objects, protocols, and categories) are compiled independently of other (related) entities. This design choice simplifies the compiler implementation. This is important for a first implementation that works as a proof-of-concept for the Logtalk language. In addition, it promotes easy development and debugging, thanks to the interactive and interpreter-like behavior of the Logtalk compiler. Entity source files can be edited and recompiled without recompilation of any related entities. However, this design choice has some disadvantages. First, both metapredicate and dynamic predicate directives are not inherited from ancestor objects. Instead, these directives must be duplicated in any entity that contains clauses for the declared predicates. Second, this design choice prevents some code optimizations and postpones the detection of some classes of errors to runtime.

8.1.6 No predefined entities

There are no predefined entities in Logtalk. This design choice was made to ensure that Logtalk remains an unbiased language. Adding predefined classes would imply adding equivalent prototypes (in terms of functionality) and vice-versa. This would complicate the language specification, cluttering it with details that belong to a Logtalk library. Thus, instead of predefined entities, the Logtalk library contains objects, protocols, and categories that implement common object-oriented functionality. These entities can be used as tutorial examples or as a starting point for developing applications. For example, the library contains objects illustrating instance creation and initialization methods similar to those found on class-based object-oriented languages.

8.1.7 One entity per source file

Each object, category, or protocol we define must be contained in its own source file. It would be easy to modify the Logtalk compiler to support compilation of several entities per source file. However, storing one entity per source file simplifies entity versioning, workgroup development of applications, and the implementation of “make”-like features in future compiler versions.

8.1.8 Distribution and use license

The current Logtalk implementation is freely available under an open source license [110]. This open source license has been chosen in order to enable widespread use (including the use in commercial applications) of the Logtalk language.

There are several reasons for this decision. Firstly, I want to avoid the problems that have impaired Prolog development and widespread use in the past due to proprietary, closed, and incompatible implementations. Secondly, this work was possible because other researchers made available the source code for their systems, allowing me to study and build on them. Thirdly, open-sourcing code and documentation means that others more skillful on coding or writing can contribute to improve the quality of the whole system. Finally, users are allowed to modify the system freely, adapting it to better suit their needs.

8.2 Implementation overview

This section provides an overview of the current Logtalk implementation, including a description on how to compile and load source files, as well as an outline of the compilation process.

8.2.1 Compiling and loading source files

In contrast with the ISO Prolog standard, Logtalk specifies a set of predicates for compiling and loading source files. These predicates work with entity file names and with an optional list of compiler options. An entity file name is the source file name without any extension or path information. As a consequence, when compiling and loading source files, the current working directory must be set to the one containing the files. This is a small price to pay to avoid all the issues with different operating systems using different conventions for path syntax.

File names

The name of an entity source file is the result of the concatenation of the entity name with the extension "`.lgt`". The Logtalk preprocessor compiles source files to Prolog code files that use, by default, a "`.pl`" extension. The documenting file that is also generated (by default) when compiling a source file, uses the extension "`.xml`". Different extensions can be set in the corresponding Prolog compiler configuration file. As an example, an object named `vehicle` should be saved in a `vehicle.lgt` source file, which will be compiled to a `vehicle.pl` Prolog file. The corresponding documenting file will be named `vehicle.xml`.

For parametric objects, the source file name should be the result of the concatenation of the entity name functor with the entity name arity and the extension "`.lgt`". For example, the parametric object `sort(_Type)` should be stored in a source file named `sort1.lgt`. This prevents file name clashes when saving Logtalk entities that have the same name but different arities.

Compiling source files to disk

The built-in predicate `logtalk_compile/1` compiles an entity or a list of entities to disk. For example:

```
| ?- logtalk_compile(tree).

| ?- logtalk_compile([listp, list]).
```

The entity compilation will use the current compiler options. The Logtalk preprocessor 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. If all the terms in a source file are valid, then there is a set of errors or potential errors that the preprocessor will try to detect and report, depending on the used compiler options. These will be described below.

Compiling and loading source files to memory

The built-in predicate `logtalk_load/1` compiles to disk and then loads to memory an entity or a list of entities.

```
| ?- logtalk_load(tree).

| ?- logtalk_load([listp, list]).
```

Compilation and loading will be performed using the current compiler options. When an entity is already loaded, the new definition replaces the old one in memory. This behavior is similar to the behavior of the reconsult predicates available in most Prolog compilers.

8.2.2 Compiler options

The Logtalk compiler supports a set of compilation options (or flags) that allows us to adapt its behavior to the particular needs of our application development.

The default values for all compiler options (except, of course, read-only options) are set on the Logtalk configuration files. Their values can be consulted or changed for the current working section using the Logtalk built-in predicates `current_logtalk_flag/2` and `set_logtalk_flag/2`. These predicates are similar to the ISO Prolog predicates `set_prolog_flag/2` and `current_prolog_flag/2`. We can also set specific options when compiling a set of entities using the Logtalk built-in predicates `logtalk_compile/2` and `logtalk_load/2`. A full description of these Logtalk built-in predicates can be found in the Appendix B.

Documenting options

The following options control the automatic generation of entity documenting files:

`xml` Controls the automatic generation of documenting files in XML format. Possible option values are `on` (the default value) and `off`.

`xsl` Sets the XSLT file to be used with the automatically generated XML documenting files. The default value is `lgtxml.xsl`.

`doctype`
: Sets the `DOCTYPE` reference in the automatically generated XML documenting files. The default value is `local`, that is, the `DOCTYPE` reference points to a local DTD file (`logtalk.dtd`). Other possible values are `standalone` (no `DOCTYPE` reference in the XML documenting files) and `web` (`DOCTYPE` reference points to a location in the Logtalk web site)[2].

The option `doctype` allows us to deal with compatibility issues when parsing a XML documenting file using a web browser or a dedicated application. Some parsers choke on the XML `DOCTYPE` declaration, some only work with local references, while others work with web references but have problems in locating a local DTD file.

Code generation options

The following two options control the code generated by Logtalk when compiling an entity:

`iso_initialization_dir`
: Controls the use of the `initialization/1` directive in the Logtalk-generated Prolog code. Possible option values are `true` (if the Prolog compiler supports the ISO definition of the directive) and `false` (if the Prolog compiler does not implement the directive or if the implementation does not conform to the ISO standard).

`code_prefix`
: Allows the definition of prefix for all functors of Logtalk-generated Prolog code. Option value must be an atom. Its default value is `''`.

The second option, `code_prefix`, is useful to solve conflicts between Logtalk-generated internal functors and user or system functors. It can also be used to take advantage of the predicate hiding features found in some Prolog compilers. For example, both YAP [68] and GNU Prolog [111] automatically hide all predicates whose names starts with a `'$'`.

Report options

When starting up, the Logtalk compiler outputs information about its version and current option values (taken from the used configuration file). During the compilation and loading of entities, the compiler outputs progress messages and may also output warnings and error messages. Error messages are always written, but the output of all the other messages and information can be controlled by using the following two options:

`startup_message`
: Controls printing of the Logtalk startup banner. Possible option values are `on` (the default value) and `off`.

`report`
: Controls the reporting of each compiled or loaded object, category, or protocol (including compilation and loading warnings). Possible option values are `on` (the default value) and `off` (silent compilation and loading).

These options are usually turned off when using Logtalk to perform batch processing.

[2] `http://www.logtalk.org/xml/1.0/logtalk.dtd`

Compilation warning options

The following options control the reporting of potential error situations:

`unknown`
: 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 default value) and `silent`.

These warnings may result from either a misspell entity name or just an entity that it will be loaded next. When using reflective class-based hierarchies (as illustrated in Chapter 1), it is not possible to define an object load order in such a way that this warning does not occur. Note that no compiler can detect all references to unknown entities: references may be constructed at runtime (similar to the construction of a metacall) or be given as an argument for some predicate.

`singletons`
: Controls the singleton variable warnings. Possible option values are `warning` (the default value) and `silent` (not recommended unless we have already checked our code and want to avoid false singletons warnings like some Prolog compilers report for variables that start with an underscore).

`named_anonymous_vars`
: Toggles the interpretation of variables that start with an underscore as named anonymous variables. Possible option values are `on` and `off` (the default value).

Singleton variables in a clause or a directive are often misspelled variables and, as such, one of the most common errors in Logtalk and Prolog programming. Often, programmers replace the anonymous variable by a *named anonymous variable*, that is, a variable which starts with an underscore, to improve code readability.

`misspelt`
: Controls the printing of warnings for misspelt calls. Possible option values are `warning` (the default value) and `silent` (not recommended).

This warning will be reported if Logtalk finds (in the body of a predicate definition) a call to a local predicate that is not defined, is not declared as dynamic, and that does not correspond to a Prolog or Logtalk built-in predicate. In most cases, these calls are mere spelling errors.

`plredef`
: Controls the printing of warnings for the redefinition of Prolog built-in predicates. Possible option values are `warning` (can be very verbose if our code redefines a lot of Prolog built-in predicates) and `silent` (the default value).

Using this option, the Logtalk compiler can be set to warn us of any redefinition of a Prolog built-in predicate inside an object or a category. Sometimes the redefinition is intentional. In other cases, we may not be aware that the used Prolog compiler may already provide the predicate as a built-in one, or we may want to ensure code portability among several Prolog compilers with different sets of built-in predicates.

`lgtredef`
: Controls the printing of warnings for the redefinition of Logtalk built-in predicates. Possible option values are `warning` (the default value) and `silent`.

Similar to the redefinition of Prolog built-in predicates, the Logtalk compiler will warn us of any redefinition of a Logtalk built-in predicate. The redefinition will probably be an error in most cases.

Portability options

Some Prolog compilers provide a large number of built-in predicates, most of which are missing in other Prolog compilers and unspecified in the Prolog ISO standard. When writing portable Logtalk applications, we can use the following option to warn us of the use of possible non-portable built-in predicates:

`portability`
: Controls the printing of warnings for calls to non-ISO specified built-in predicates. Possible option values are `warning` and `silent` (the default value).

Whenever possible, we should encapsulate all non-portable code in a small set of objects with clearly-defined protocols. We can then turn this option off during the compilation of those objects.

Compilation options

The following option toggles a limited "make"-like feature of the Logtalk compiler:

`smart_compilation`
: Controls the use of smart compilation of source files to avoid recompiling files that remain unchanged since the last time they were compiled. Possible option values are `on` and `off` (the default value).

However, this feature is not supported by all Prolog compilers. Some of them lack an operating system interface that gives access to a file last modification time. This option should be turned off when switching Prolog compilers or operating systems in order to prevent compatibility problems with the Prolog files that result from the compilation of Logtalk entities.

Other options

The following option can be used by programs that need to check whether some version-dependent feature is available:

`version`
: Read-only option that stores the current Logtalk version.

Logtalk version numbers use the format `major.minor.patch`. For example, Logtalk version `2.15.2` should be read as edition `2`, release `15`, patch `2`. The edition number denotes the current generation of the Logtalk language. The release number is incremented whenever a new feature is implemented or significant changes to existing features are made. The patch number is incremented when minor fixes are made.

8.2.3 Compiler and runtime error handling

Logtalk error (or exception) handling is performed by using the ISO Prolog predicates `catch/3` and `throw/1`. All exceptions thrown terms are modeled after the ISO Prolog standard. The goal is to provide close integration with the standard and to minimize the learning curve for new Logtalk users with a Prolog programming background.

A complete and detailed description of runtime errors that can result from misuse of the Logtalk built-in predicates and built-in methods can the found in the Appendix B.

8.2.4 Parsing and translating source files

Compilation of a Logtalk source file to Prolog code is performed in three steps:

1. The preprocessor opens the source file and reads it, term by term (each term is either a directive or a clause). With the operator definitions set by the compiler, all terms of a syntactically correct Logtalk source file are also valid Prolog terms. These terms are compiled to memory using a set of dynamic predicates for storing the compilation results.

2. The results of the first step are checked for errors and for calls to any Logtalk or Prolog built-in predicate that might have been redefined.

3. The compilation result is written to disk as a Prolog code file and all the intermediate compilation predicates are retracted. To load into memory a successfully compiled entity, this Prolog code file is then compiled and loaded by the chosen Prolog compiler.

The following sections will describe in detail the compilation of entities, entity relations, entity directives, predicate directives, and predicate clauses.

8.3 Identifiers, prefixes, functors, and tables

The implementation of Logtalk as a preprocessor, which compiles Logtalk code to Prolog code, implies that we must construct new identifiers and functors to represent the compiled code. The basic idea is to construct a prefix from an entity identifier, and then to use this prefix to construct all the internal functors needed to represent the compiled entity. Entity prefixes are accessed through entity tables that map entity identifiers to entity prefixes. Entity prefixes are then used to access the internal functors used in all bookkeeping tables, including the tables of declared and defined predicates, of each entity.

In order to make our description on compiling objects, categories and protocols more clear, we will use the following example object in the next sections:

```
:- object(stack).

    :- public(top/1).
    :- public(push/1).
    :- public(pop/1).
    :- public(empty/0).
```

```
    :- public(map/3).
    :- metapredicate(map(*, ::, *)).

    :- private(stack_/1).
    :- dynamic(stack_/1).

    top(Top) :-
        ...

:- end_object.
```

8.3.1 Entity prefix

The entity identifier is used to construct an atom, the *entity prefix*, which will be used as a prefix for all functors we need when compiling the entity. This prefix is the result of the concatenation of the entity functor with the entity arity followed by an underscore. For example, if we are compiling an object named `stack`, then the corresponding prefix will be the atom `stack0_`. Because all entities share the same namespace, all identifiers must be unique and, consequently, by construction, all prefixes and thus all derived functors will also be unique. Note that it is always possible, but unlikely, that a user predicate will conflict with a predicate generated by Logtalk when compiling an entity. Logtalk supports a compiler option (named `code_prefix`) for setting an arbitrary prefix to all functors generated when compiling an entity. An alternative solution will be a portable mechanism for hiding private system predicates, similar to those found in some Prolog compilers.

8.3.2 Entity tables

The runtime engine maintains a set of tables mapping entity identifiers to the corresponding prefixes. These tables are not strictly necessary but they do avoid the performance penalties of repeatedly constructing prefixes when processing messages. For objects, we use the following table:

```
lgt_current_object_(Object, Prefix, Dcl, Def, Super).
```

Each table entry contains the entity identifier, the entity prefix, and three additional arguments, which are cached functors that will be described later. These functors are the most used ones when processing a message. They are cached in this table in order to improve message sending performance. For our example object `stack`, the corresponding table entry will be:

```
lgt_current_object_(stack,
        stack0_, stack0__dcl, stack0__def, stack0__super).
```

For protocols and categories, we use the following two tables:

```
lgt_current_protocol_(Protocol, Prefix).

lgt_current_category_(Category, Prefix).
```

By convention, a Logtalk compiler functor ending with an underscore denotes a dynamic predicate. When an entity is compiled and loaded, an entry is added to the corresponding table. Conversely, abolishing an entity removes the corresponding table entry.

8.3.3 Predicate tables

What information do we need to represent when compiling an entity? First, we need to represent each predicate declaration. Thus, the compiled code will contain a table of all predicate declarations. The functor of this table is the result of the concatenation of the entity prefix with the atom `_dcl` (interpreted as an abbreviation of the word "declaration"). Therefore, for our example object `stack`, we will have the table:

```
stack0__dcl(...).
```

Second, for entities that may contain predicate definitions, we need a way to go from an entity identifier plus a predicate term to the compiled predicate term. The table representing the mapping from user predicate names to the corresponding compiled names uses a functor constructed by appending the entity prefix to the atom `_def` (for definition):

```
stack0__def(...).
```

Not all predicate declarations and definitions exist at the time of entity compilation. Logtalk objects support the dynamic declaration and definition of predicates. Dynamically declared predicates can be abolished at runtime, unlike predicates declared at compilation time[3]. For this purpose we use two additional tables whose functors result from appending the entity prefix to the atoms `_ddcl` (for dynamic declarations) and `_ddef` (for dynamic definitions):

```
stack0__ddcl(...).

stack0__ddef(...).
```

Runtime use of these four tables implies the construction of the corresponding call using the ISO Prolog built-in predicate `=../2`. When processing a message, we must first check its validity against the predicate declaration tables, and then find a predicate declaration to answer it using the predicate definition tables. However, by design, and in order to improve performance, only two such calls need to be constructed per message. The details of these tables will be further discussed in the next section.

8.3.4 Linking clauses

Next, we need to represent inheritance relations so that we can perform efficient lookups of both predicate declarations and predicate definitions. This is accomplished by generating *linking clauses* that connect an entity declaration and definition tables with the corresponding entity ancestor tables. The meaning of an entity ancestor depends on whether we are compiling objects, categories, or protocols. Moreover, in the case of

[3]Note that the concept of *dynamic predicate declaration* is orthogonal to the concept of *dynamic predicate*. Predicates declared in an entity source file are *statically* declared and cannot be abolished, even if some of them are declared as dynamic. In the case of a source file containing a definition for a dynamic entity, its predicates can only be abolished by abolishing the entity itself.

objects, it depends on whether we are compiling prototypes, instances, or classes. Note that, for any compiled entity, the linking clauses connecting to the local predicate tables are always generated, even for *stand-alone* entities.

All linking clauses are static as a consequence of the design choice that entity relations are static properties, not updatable at runtime. How these linking clauses are generated will be discussed later on this chapter.

8.3.5 Entity functors clause

How do we go from an entity identifier to each functor? First, we use the entity tables that map entity identifiers to entity prefixes. Then we use a single clause predicate, the *entity functors clause*, whose functor is the entity prefix, which contains all the functors as arguments. For objects, this clause has the following format:

```
Prefix(Dcl, Def, Super, IDcl, IDef, DDcl, DDef).
```

The clause arguments are the functors described previously in this section. For our example object `stack` the functors clause will be:

```
stack0_(stack0__dcl, stack0__def,
        stack0__super,
        stack0__idcl, stack0__idef,
        stack0__ddcl, stack0__ddef).
```

For categories, we need only two functors, one for predicate declarations, and another for predicate definitions:

```
Prefix(Dcl, Def).
```

For protocols, taking into account that they can only contain predicate declarations, we need only one functor:

```
Prefix(Dcl).
```

In theory, runtime use of these clauses would imply the construction of the corresponding call using the ISO Prolog built-in predicate `=../2`. However, in order to improve performance, the most used functors in message processing are cached in the object identifier table.

8.4 Compiling predicate directives

As explained before on this chapter, when compiling entity predicate directives, Logtalk constructs a static table of predicate declarations. At runtime, new predicate declarations can be asserted into an object, resulting in a dynamic table of predicate declarations. Note that when we talk about a static or dynamic table of predicate declarations we are talking about the table itself, not about static or dynamic predicates. Only the scope, dynamic, and metapredicate directives are used in the construction of these predicate declaration tables. Discontiguous predicate directives are compiled by generating the equivalent Prolog directives for the compiled predicates. Predicate documenting directives are only used in the automatic generation of documenting files.

8.4.1 Static table of predicate declarations

Each entity contains a table of all predicates declared at compilation time, the *static table of predicate declarations*. For each predicate, a table entry with the following format is generated:

```
Dcl(Pred, Scope, Type, Meta).
```

The value of each argument comes from the scope, dynamic, and metapredicate directives describing the predicate. The value of the first argument, `Pred`, is a predicate term (`Functor(...)`), not a predicate indicator (`Functor/Arity`), for performance reasons. The second argument, `Scope`, represents the predicate scope using one of the following terms:

`p(p(p))`
: public predicate

`p(p)`
: protected predicate

`p`
: private predicate

This representation allows checking for public or protected predicates with a single unification call: the term `p(_)` unifies with both the terms representing public scope and protected scope. This check needs to be performed every time a message to *self* is processed.

The value of the third argument, `Type`, can either be the atom `static` or the atom `dynamic`. All predicates are static unless explicitly declared dynamic.

The value of the last argument, `Meta`, can either be the atom `no` (meaning that the predicate is not a meta-predicate) or a meta-predicate term. For the example object `stack` the clauses generated for the declared predicates will be:

```
stack0__dcl(top(_), p(p(p)), static, no).
stack0__dcl(push(_), p(p(p)), static, no).
stack0__dcl(pop(_), p(p(p)), static, no).
stack0__dcl(empty, p(p(p)), static, no).
stack0__dcl(map(_,_,_), p(p(p)), static, map(*, ::, *)).
stack0__dcl(stack_(_), p, dynamic, no).
```

If there are no (local) predicate declarations, then the following catchall clause is generated:

```
Dcl(_, _, _, _) :-
    fail.
```

This clause ensures that there will be no unknown predicate runtime errors when looking up a predicate declaration.

8.4.2 Dynamic table of predicate declarations

A second predicate declaration table is used for predicates that are dynamically declared at runtime (by asserting a clause for them into an object). This table is a short version

of the static table of predicate declarations described in the previous section but uses a different functor:

```
DDcl(Pred, Scope).
```

The value of the second argument, `Scope`, depends on the object receiving the asserting message, as explained in Chapter 3. Note that there is no need for a `Type` argument, as the predicates are always dynamic. In addition, since Logtalk does not support the dynamic declaration of metapredicates, the fourth argument of the static table, `Meta`, is not necessary. For example, the message:

```
| ?- stack::assertz(foo(1)).
```

would result in the following table entry:

```
stack0__ddcl(foo(_), p(p(p))).
```

This table is declared as a dynamic predicate, so there is no need for a catchall clause in order to prevent errors when there are no table entries.

8.5 Compiling predicate clauses

Compilation of predicate clauses is performed in two stages. In the first stage, all predicate clauses are compiled one by one. Each predicate clause is compiled by first compiling its head and then its body. In the second stage, we update the compiled clauses to reflect any redefined built-in predicate. Definite clause grammar rules are first converted to predicate clauses and then compiled as any other clause.

8.5.1 Compiling clause heads

Clause heads are compiled by constructing a new internal functor from the predicate functor and by appending three extra arguments that are used for context information passing.

Predicate prefix

The predicate internal functor is constructed by concatenating the entity prefix with the predicate prefix. The predicate prefix is the result of the concatenation of the predicate functor with the predicate arity. For example, for the predicate `top/1`, declared in the object `stack`, the internal predicate functor will be `stack0_top1`.

Execution context

The compiled clause heads contain three extra arguments, appended to the existing ones; these arguments are used at runtime to pass the execution context. For example, the predicate `top/1` mentioned above will be translated to the following clause:

```
stack0_top1(Top, Sender, This, Self) :-
    ...
```

The context arguments are instantiated at runtime, when a clause is selected to answer a message. The first context argument, `Sender`, is instantiated with the sender of the message. The second context argument, `This`, is instantiated with the entity containing the predicate definition. The third context argument, `Self`, is instantiated with the object that has received the message. Note that, by adding the context arguments after the existing ones, we preserve any use of first-argument indexing the programmer might have employed to optimize predicate calls.

8.5.2 Predicate definition tables

As explained before, Logtalk constructs a static table of predicate definitions when compiling entity predicate clauses. At runtime, new predicate clauses can be asserted into an object for predicates that are not defined at the time of the object compilation, resulting in a dynamic table of predicate definitions.

Static table of predicate definitions

Compiling an object or a category generates a *static table of predicate definitions*. This table maps the user predicate names to the internal ones for all predicates that are defined at compilation time. The table entry template is:

```
Def(Pred, Sender, This, Self, Call).
```

The first argument is a predicate term, not a predicate indicator. For the example above, the corresponding table entry will be:

```
stack0__def(top(Stack, Top), Sender, This, Self,
            stack0_top2(Stack, Top, Sender, This, Self)).
```

In the case of objects, a table entry is also generated for each contained dynamic predicate directive, even if the object contains no clauses for the corresponding predicates at compilation time. Note that clauses for dynamic predicates can always be asserted at runtime.

For categories that do not contain any predicate definitions, and for objects that do not contain either predicate definitions or dynamic predicate directives, the following catchall clause is generated:

```
Def(_, _, _, _, _) :-
    fail.
```

This clause ensures that there will be no unknown predicate runtime errors when looking up a predicate definition. Assume, for example, that an object imports a category that does not contain any predicate definitions. Nevertheless, there will be a linking clause enabling the object to search for predicate definitions inside the category. This linking clause is always generated because the category may be later updated by adding predicate definitions, thus updating the importing object without recompiling it.

Dynamic table of predicate definitions

There is a second predicate definition table, the *dynamic table of predicate definitions*, which is used for definitions of predicates that are dynamically declared (by asserting a

clause for them into an object) or whose declarations are inherited from another entity (in this case, the predicates must have been declared dynamic). This table is generated and updated at runtime. It is similar to the static table of predicate definitions described in the previous section but uses a different functor:

```
DDef(Pred, Sender, This, Self, Call).
```

The table entry arguments have the same meaning as the arguments of the table described in the previous section. This table is declared as a dynamic predicate, so there is no need for a catchall clause in order to prevent errors when there are no defined table entries.

Note that this table exists only for objects (we cannot assert new predicate definitions in a category: categories can only contain static predicates and, as such, only need a static table predicate definitions).

8.5.3 Compiling clause bodies

A clause body is compiled by compiling its goals. The goals can be arguments of control constructs, calls to Prolog and Logtalk built-in predicates, local predicates, message sending calls, and metacalls. As an object or category may redefine both Prolog and Logtalk built-in predicates, all clauses containing a call to a redefined built-in predicate are compiled such that the new definition will be called instead of the predefined one.

The compilation is performed in three stages. First, each call in the clause body is compiled as described below. Second, the result of the first stage is updated to apply any redefinition of built-in predicates. Third, the compiled body is simplified by removing any redundant calls to the built-in predicate `true`.

Compiling built-in metapredicate calls

Calls to Prolog and Logtalk built-in metapredicates are compiled by first compiling the corresponding metacalls and then by compiling the metapredicate call as any other built-in predicate, as explained below.

Compiling built-in predicate calls

Calls to Logtalk built-in predicates are translated into calls to the Logtalk internal predicates implementing them. Calls to Prolog built-in predicates are copied without any further processing.

Compiling built-in method calls

Calls to the built-in message execution context methods are translated by unifying the method argument with the corresponding context information argument in the extended clause head. Thus, as the unification is performed at compilation time, the runtime performance cost of these methods is null.

Calls to other Logtalk built-in methods are translated into calls to the Logtalk internal predicates that implement them.

Compiling message sending calls

Logtalk design choice of dynamic binding for message processing greatly simplifies the compilation of message sending calls. The preprocessor first checks, if possible, whether the target object and the message are valid. Then, it translates the call to a call of the internal predicate that implements the runtime message processing.

Compiling metacalls

When compiling a metapredicate, all arguments in the clause head that are metavariables are collected into a list that is used in the compilation of the clause body. The list of metavariables is constructed from the corresponding metapredicate directive. When compiling the clause body, the argument of each metacall is checked against the list of metavariables. When the metacall argument is a metavariable, the metacall is compiled so that it is executed in the context of the *sender*; otherwise it is compiled so that it is executed in the context of *this*.

Compiling local predicate calls

Calls to locally defined predicates are simply compiled by translating the call into a call to the internal form of the predicate.

8.6 Compiling entity relations

Now that I have described how predicate directives and predicate clauses are compiled, I need to detail how the relations between entities are compiled. As explained before, the basic idea is to generate a set of linking clauses that will connect the predicate declaration and definition tables of an entity to the same tables in the related entities. These clauses make use of the same functors of the predicate declaration and definition tables, with one or two extra arguments appended at the end of the clause heads. These extra arguments are used to return the entity containing the inherited declaration or the inherited definition. They are necessary to check scope validity and to support the Logtalk built-in reflection method `predicate_property/2`.

Linking clauses ordering

We need to establish suitable orderings for entity linking clauses in order to solve conflicts between inherited predicate declarations and inherited predicate definitions.

Protocols When a protocol extends other protocols, the order of the linking clauses for this relation is the same as the order of the extended protocols in the protocol-opening directive.

Categories When a category implements some protocols, the order of the linking clauses for this relation is the same as the order of the implemented protocols in the category-opening directive.

Prototypes A prototype may implement one or more protocols, import one or more categories, and extend one or more prototypes. The ordering of linking clauses, for each relation type, is the same as the order of the entities in the object-opening directive. Considering the ordering of the linking clauses between relation types, we have: first, implemented protocols; second, imported categories; third, extended objects.

Instances/classes Non-prototype objects (i.e. instances or classes) may implement one or more protocols, import one or more categories, instantiate one or more classes, and specialize one or more classes. The ordering of linking clauses, for each relation type, is the same as the order of the entities in the object-opening directive. For the ordering of the linking clauses between relation types, we have: first, implemented protocols; second, imported categories; third, relations with other objects.

Scope container versus true container

For objects, we need to make a distinction, when inheriting predicate declarations, between the *scope container* and the *true container*. This distinction is necessary due to the semantics of protocols and categories. If an object implements (imports) a predicate declaration from a protocol (category), the true container will be the protocol (category), but the scope container will be the object implementing (importing) the protocol (category). The scope container, as its name suggests, is used to check scope constraints when processing messages, while the true container is returned by the built-in reflection method `predicate_property/2`.

8.6.1 Compiling protocol relations

Protocols may either be *stand-alone* entities or be defined as extensions of other protocols.

Protocols contain predicate declarations (but not predicate definitions) that can be implemented by any object or category. The predicate called when looking up a predicate declaration uses the same functor as the predicate declaration table but with an additional argument used to return the protocol containing the declaration. The first linking clause connects to the local table of predicate declarations:

```
Dcl(Pred, Scope, Type, Meta, Protocol) :-
    Dcl(Pred, Scope, Type, Meta).
```

Next, we have a linking clause for each extended protocol with the format:

```
Dcl(Pred, Scope, Type, Meta, Container) :-
    ExtPtcDcl(Pred, Scope, Type, Meta, Container).
```

The order of these clauses is the same as the order of the extended protocols in the protocol-opening directive.

As an example, a protocol-opening directive such as:

```
:- protocol(extd_listp,
    extends(basic_listp)).
```

results in the following set of linking clauses:

```
extd_listp0__dcl(Pred, Scope, Type, Meta, extd_listp) :-
    extd_listp0__dcl(Pred, Scope, Type, Meta).

extd_listp0__dcl(Pred, Scope, Type, Meta, Container) :-
    basic_listp0__dcl(Pred, Scope, Type, Meta, Container).
```

8.6.2 Compiling category relations

Categories may either be *stand-alone* entities or implement one or more protocols.

Categories contain predicate declarations and definitions that can be imported by any number of objects. Since protocols can only contain predicate declarations, we only need linking clauses connecting a category predicate declaration table to the tables in each implemented protocol. A category cannot inherit predicate definitions from other entities. Category linking clauses are thus similar to protocol linking clauses.

The first linking clause connects a category to its local predicate declaration table:

```
Dcl(Pred, Scope, Type, Meta, Category) :-
    Dcl(Pred, Scope, Type, Meta).
```

After this clause, we will have a linking clause for each implemented protocol:

```
Dcl(Pred, Scope, Type, Meta, Container) :-
    PtcDcl(Pred, Scope, Type, Meta, Container).
```

The order of these clauses is the same as the order of the implemented protocols declaration in the category-opening directive.

As an example, a category-opening directive such as:

```
:- category(category,
    implements(protocol)).
```

results in the following set of linking clauses:

```
category0__dcl(Pred, Scope, Type, Meta, category) :-
    category0__dcl(Pred, Scope, Type, Meta).

category0__dcl(Pred, Scope, Type, Meta, Container) :-
    protocol0__dcl(Pred, Scope, Type, Meta, Container).
```

8.6.3 Compiling prototype relations

A prototype can be a *stand-alone* entity or it may extend other prototypes, implement some protocols, or import some categories. Therefore, compiling a predicate results in two sets of linking clauses: one set for predicate declarations and another set for predicate definitions.

Linking clauses for predicate declarations

A predicate declaration may be local, contained in an implemented protocol, contained in an imported category, or found in some ancestor object. Therefore, the first two

linking clauses connect to the local (static and dynamic) predicate declaration tables:

```
Dcl(Pred, Scope, Type, Meta, Object, Object) :-
    Dcl(Pred, Scope, Type, Meta).

Dcl(Pred, Scope, (dynamic), no, Object, Object) :-
    DDcl(Pred, Scope).
```

Second, we have a linking clause for each (if any) implemented protocol:

```
Dcl(Pred, Scope, Type, Meta, Object, TrueCtn) :-
    ProtocolDcl(Pred, Scope, Type, Meta, TrueCtn).
```

Third, we have a linking clause for each (if any) imported category:

```
Dcl(Pred, Scope, Type, Meta, Object, TrueCtn) :-
    CategoryDcl(Pred, Scope, Type, Meta, TrueCtn).
```

Finally, we have a linking clause for each (if any) extended (parent) prototype:

```
Dcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn) :-
    ParentDcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn).
```

The order of these clauses is the same as the order of the extended objects in the object-opening directive.

As an example, an object-opening directive such as:

```
:- object(proto,
    implements(protocol),
    imports(category),
    extends(parent)).
```

results in the following set of linking clauses for predicate declarations:

```
proto0__dcl(Pred, Scope, Type, Meta, proto, proto) :-
    proto0__dcl(Pred, Scope, Type, Meta).

proto0__dcl(Pred, Scope, dynamic, no, proto, proto) :-
    proto0__ddcl(Pred, Scope).

proto0__dcl(Pred, Scope, Type, Meta, proto, TrueCtn) :-
    protocol0__dcl(Pred, Scope, Type, Meta, TrueCtn).

proto0__dcl(Pred, Scope, Type, Meta, proto, TrueCtn) :-
    category0__dcl(Pred, Scope, Type, Meta, TrueCtn).

proto0__dcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn) :-
    parent0__dcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn).
```

Thus, the simplicity of prototype hierarchies translates into a simple set of linking clauses. As we will see in the next section, the linking clauses for classes and instances are significantly more complex due to the distinction between instantiation and specialization relations.

Linking clauses for predicate definitions

A predicate definition may be local, imported from a category, or found in some ancestor object. Thus, the linking clauses for predicate definitions are similar to the linking clauses for predicate declarations described above. The first two clauses for the definition lookup predicate provide access to the local predicate definition tables:

```
Def(Pred, Sender, This, Self, Call, Object) :-
    Def(Pred, Sender, This, Self, Call).

Def(Pred, Sender, This, Self, Call, Object) :-
    DDef(Pred, Sender, This, Self, Call).
```

Next, we have a linking clause for each imported category:

```
Def(Pred, Sender, Object, Self, Call, Category) :-
    CategoryDef(Pred, Sender, Object, Self, Call).
```

Finally, we have a linking clause for each parent prototype:

```
Def(Pred, Sender, Object, Self, Call, Container) :-
    ParentDef(Pred, Sender, Parent, Self, Call, Container).
```

For the object-opening directive presented above, the linking clauses will be:

```
proto0__def(Pred, Sender, This, Self, Call, proto) :-
    proto0__def(Pred, Sender, This, Self, Call).

proto0__def(Pred, Sender, This, Self, Call, proto) :-
    proto0__ddef(Pred, Sender, This, Self, Call).

proto0__def(Pred, Sender, This, Self, Call, category) :-
    category0__def(Pred, Sender, This, Self, Call).

proto0__def(Pred, Sender, Object, Self, Call, Container) :-
    parent0__def(Pred, Sender, Object, Self, Call, Container).
```

Once again, the simplicity of prototype hierarchies translates into a simple set of linking clauses.

Linking clauses for super calls

Whenever we execute a *super* call, the lookup procedure for the predicate definition uses a set of linking clauses that uses a functor constructed by concatenating the entity prefix with the atom `_super`.

When compiling a root prototype, that is, a prototype that does not extend other prototypes, the following catchall clause is generated in order to prevent "predicate not found" errors at runtime:

```
Super(_, _, _, _, _, _) :-
    fail.
```

If our prototype extends other prototypes, then we generate a linking clause for each extended prototype:

```
Super(Pred, Sender, Object, Self, Call, Container) :-
    ParentDef(Pred, Sender, Parent, Self, Call, Container).
```

The order of these clauses is the same as the order of the parents in the prototype-opening directive.

For our example, the linking clause will be:

```
proto0__super(Pred, Sender, proto, Self, Call, Container) :-
    parent0__def(Pred, Sender, parent, Self, Call, Container).
```

8.6.4 Compiling instantiation and specialization relations

An object may instantiate one or more classes, and may also be instantiated by other objects. At the same time, an object may also implement some protocols, import some categories and specialize some objects (that will play the role of superclasses), or be specialized by other objects (that will play the role of subclasses).

For predicate declarations, the lookup always starts at the instance classes. However, for predicate definitions, the lookup starts at the instance itself and then, if it fails, proceeds to the instance classes and then to the class superclasses. We also need linking clauses to represent the transitive nature of the specialization relation. These transitive clauses use two functors constructed by concatenating an entity prefix with the atoms `_idcl` (for inherited declarations) and `_idef` (for inherited definitions).

Linking clauses for predicate declarations

For an instance, the linking clauses for predicate declarations simply redirect the lookup of declarations to the instance classes. That is, we will have a linking clause for each instantiated class:

```
Dcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn) :-
    ClassIDcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn).
```

If an object does not instantiates any class, then a catchall clause is generated:

```
Dcl(_, _, _, _, _, _) :-
    fail.
```

An object can also play the role of a class, by being instantiated or specialized by other objects. When looking up predicate declarations in an object (playing the role of a class) coming from a descendant object, we use a set of linking clauses using the `IDcl` functor. The first two clauses connect to the local tables of predicate declarations:

```
IDcl(Pred, Scope, Type, Meta, Object, Object) :-
    Dcl(Pred, Scope, Type, Meta).

IDcl(Pred, Scope, (dynamic), no, Object, Object) :-
    DDcl(Pred, Scope).
```

Second, we have a linking clause for each implemented protocol:

```
IDcl(Pred, Scope, Type, Meta, Object, TrueCtn) :-
    ProtocolDcl(Pred, Scope, Type, Meta, TrueCtn).
```

Third, we have a linking clause for each imported category:

```
IDcl(Pred, Scope, Type, Meta, Object, TrueCtn) :-
    CategoryDcl(Pred, Scope, Type, Meta, TrueCtn).
```

Last, we have a linking clause for each specialized class:

```
IDcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn) :-
    SuperclassIDcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn).
```

As an example, an object-opening directive such as:

```
:- object(class,
    implements(protocol),
    imports(category),
    instantiates(meta),
    specializes(super)).
```

results in the following set of linking clauses for predicate declarations:

```
class0__dcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn) :-
    meta0__idcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn).

class0__idcl(Pred, Scope, Type, Meta, class, class) :-
    class0__dcl(Pred, Scope, Type, Meta).

class0__idcl(Pred, Scope, dynamic, no, class, class) :-
    class0__ddcl(Pred, Scope).

class0__idcl(Pred, Scope, Type, Meta, class, TrueCtn) :-
    protocol0__dcl(Pred, Scope, Type, Meta, TrueCtn).

class0__idcl(Pred, Scope, Type, Meta, class, TrueCtn) :-
    category0__dcl(Pred, Scope, Type, Meta, TrueCtn).

class0__idcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn) :-
    super0__idcl(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn).
```

Linking clauses for predicate definitions

A predicate definition can be local, imported from a category, or inherited from one of the instance classes. Therefore, first we have two clauses connecting to the local tables of predicate definitions:

```
Def(Pred, Sender, This, Self, Call, Object) :-
    Def(Pred, Sender, This, Self, Call).

Def(Pred, Sender, This, Self, Call, Object) :-
    DDef(Pred, Sender, This, Self, Call).
```

Next, we have a linking clause for each imported category:

```
Def(Pred, Sender, Object, Self, Call, Category) :-
    CategoryDef(Pred, Sender, Object, Self, Call).
```

Finally, we have a linking clause for each instantiated class:

```
Def(Pred, Sender, Object, Self, Call, Container) :-
    ClassIDef(Pred, Sender, Class, Self, Call, Container).
```

When looking up predicate definitions in an object (playing the role of a class) coming from a descendant object we use a set of linking clauses using the `IDef` functor. The first two clauses connect to the local tables of predicate definitions:

```
IDef(Pred, Sender, This, Self, Call, Object) :-
    Def(Pred, Sender, This, Self, Call).

IDef(Pred, Sender, This, Self, Call, Object) :-
    DDef(Pred, Sender, This, Self, Call).
```

Next, we have a linking clause for each imported category:

```
IDef(Pred, Sender, Object, Self, Call, Category) :-
    CategoryDef(Pred, Sender, Object, Self, Call).
```

Finally, we have a linking clause for each specialized class:

```
IDef(Pred, Sender, Class, Self, Call, Container) :-
    SuperclassIDef(Pred, Sender, Super, Self, Call, Container).
```

Note that, although the set of linking clauses for the `Def` functor is similar to the set of clauses for the `IDef` functor, there is a crucial difference: when we send a message to an object, the search for a predicate definition starts at the object itself and then (if not found) it proceeds to the object classes. When we search an object playing the role of a class for a predicate definition, the search continues at the class superclasses, not at the class metaclasses.

For our object-opening directive example, the linking clauses will be:

```
class0__def(Pred, Sender, This, Self, Call, class) :-
    class0__def(Pred, Sender, This, Self, Call).

class0__def(Pred, Sender, This, Self, Call, class) :-
    class0__ddef(Pred, Sender, This, Self, Call).

class0__def(Pred, Sender, class, Self, Call, category) :-
    category0__def(Pred, Sender, class, Self, Call).

class0__def(Pred, Sender, class, Self, Call, Container) :-
    meta0__idef(Pred, Sender, meta, Self, Call, Container).

class0__idef(Pred, Sender, This, Self, Call, class) :-
    class0__def(Pred, Sender, This, Self, Call).
```

```
class0__idef(Pred, Sender, This, Self, Call, class) :-
    class0__ddef(Pred, Sender, This, Self, Call).

class0__idef(Pred, Sender, class, Self, Call, category) :-
    category0__def(Pred, Sender, class, Self, Call).

class0__idef(Pred, Sender, class, Self, Call, Container) :-
    super0__idef(Pred, Sender, super, Self, Call, Container).
```

Linking clauses for super calls

In class-based hierarchies, a predicate defined in a class can be specialized both in the descendant subclasses and in the descendant instances. In the first case, the following linking clause will be generated:

```
Super(Pred, Sender, Class, Self, Call, Container) :-
    SuperclassIDef(Pred, Sender, Super, Self, Call, Container).
```

In the second case, the linking clause will be:

```
Super(Pred, Sender, Object, Object, Call, Container) :-
    ClassIDef(Pred, Sender, Class, Object, Call, Container).
```

For our example, the linking clauses will be:

```
class0__super(Pred, Sender, class, class, Call, Container) :-
    meta0__idef(Pred, Sender, meta, class, Call, Container).

class0__super(Pred, Sender, class, Self, Call, Container) :-
    super0__idef(Pred, Sender, super, Self, Call, Container).
```

At first glance, the clauses above seem to imply that, when an object both instantiates and specializes other objects, a *super* call would result in a search along two inheritance links. However, note that the first clause will only be used if the value of the *This* argument is the same as the *Self* argument, allowing us to distinguish between an instance specialized method and a subclass specialized method. One problem of this solution is that a backtracking choice point will be always created when processing a *super* call. However, the choice point will be removed before calling the specialized definition.

8.6.5 Compiling protected and private relations

All linking clauses described so far have assumed a public relation between entities. As explained in previous chapters, Logtalk also supports protected and private relations. All relation types can be qualified using the keywords `protected` and `private`. The linking clauses for protected and private relations are derived from the public relation clauses by ensuring the conversion of the scope of *inherited* predicates.

Protocol extension relations

For a protected relation, all public predicates of the extended protocol will be inherited as protected. Protected and private predicates will retain their scope. The format for the linking clause will be:

```
Dcl1(Pred, Scope, Type, Meta, Object, Container) :-
    Dcl2(Pred, Scope2, Type, Meta, Container),
    (Scope2 == p -> Scope = p; Scope = p(p)).
```

Note that the test in the conditional goal uses the Prolog comparison predicate `==/2`, instead of the unification predicate `=/2`. This enables optimization of the conditional call in several Prolog compilers.

For a private relation, the corresponding linking clause will be simpler as all predicates of the extended protocol will be inherited as private:

```
Dcl1(Pred, p, Type, Meta, Obj, Container) :-
    Dcl2(Pred, _, Type, Meta, Container).
```

As an example, given a protocol-opening directive such as:

```
:- protocol(icecream,
    extends(private::vanilla, protected::chocolate)).
```

the following two linking clauses will be generated:

```
icecream0__dcl(Pred, p, Type, Meta, Container) :-
    vanilla0__dcl(Pred, _, Type, Meta, Container).

icecream0__dcl(Pred, Scope, Type, Meta, Container) :-
    chocolate0__dcl(Pred, Scope2, Type, Meta, Container),
    (Scope2 == p -> Scope = p; Scope = p(p)).
```

Note that a protocol hierarchy is always transparent for an object implementing it. The example above will work in the same way as if we collect all predicate declarations of the three protocols (`icecream`, `vanilla`, and `chocolate`) into a single protocol.

Implementation and importation relations

The linking clauses for protocol implementation and category importation are similar to the protocol extension clauses presented above. For example, an object-opening directive such as:

```
:- object(root,
    implements(private::protocol),
    imports(protected::category)).
```

will result in the following two linking clauses:

```
root0__dcl(Pred, p, Type, Meta, root, Container) :-
    protocol0__dcl(Pred, _, Type, Meta, Container).

root0__dcl(Pred, Scope, Type, Meta, root, Container) :-
    category0__dcl(Pred, Scope2, Type, Meta, Container),
    (Scope2 == p -> Scope = p; Scope = p(p)).
```

As always, the scope container is the object implementing the protocol or importing the category.

Extension, instantiation, and specialization relations

For extension, instantiation, and specialization relations, the linking clauses must take into account the distinction between the scope container and the true container of the inherited predicates.

The linking clause for a protected inheritance is similar to the linking clause for other relations:

```
Dcl1(Pred, Scope, Type, Meta, ScopeCtn, TrueCtn) :-
    Dcl2(Pred, Scope2, Type, Meta, ScopeCtn, TrueCtn),
    (Scope2 == p -> Scope = p; Scope = p(p)).
```

However, the linking clause for private inheritance is more complex than for the linking clauses for public and protected relations:

```
Dcl1(Pred, p, Type, Meta, ScopeCtn, TrueCtn) :-
    Dcl2(Pred, Scope2, Type, Meta, ScopeCtn2, TrueCtn),
    (Scope2 == p -> ScopeCtn = ScopeCtn2; ScopeCtn = Object1).
```

The conditional call is required for two reasons. First, it ensures that inherited public and protected predicates will act as private predicates for the object performing the private inheritance. Second, it prevents ancestor private predicates from being called from the descendant object. As an example, for an object-opening directive such as:

```
:- object(proto,
    extends(private::parent)).
```

the following linking clause will be generated:

```
proto0__dcl(Pred, p, Type, Meta, ScopeCtn, TrueCtn) :-
    parent0__dcl(Pred, Scope2, Type, Meta, ScopeCtn2, TrueCtn),
    (Scope2 == p -> ScopeCtn = ScopeCtn2; ScopeCtn = proto).
```

For public and protected predicates declared in the parent prototype, `parent`, the scope is set to private and the *scope container* is set to the object inheriting the predicates, `proto`. Thus, when checking scope rules, those predicates will act as private predicates declared in the object `proto`. For private predicates, in order not to break object encapsulation, the *scope container* is passed unchanged.

8.7 Runtime support for events and monitors

In Logtalk, for each message sent using the `::/2` control construct, the runtime engine must check if the corresponding events are being monitored. Therefore, the performance of this check is crucial to the performance of the whole system. Ideally, event checking should be performed in constant time and independently of the number of spied events. This is essential to ensure that the existence of monitored objects does not affect the processing of messages sent to the remaining objects. This performance goal can be attained on Prolog compilers which support first argument indexing for dynamic code.

On these compilers, the event representation takes advantage of first argument indexing to ensure that — in general case — event checking is performed in constant time.

The Logtalk runtime engine uses two dynamic tables for storing events and the corresponding monitors. The structure of these tables is as follows:

```
lgt_before_(Object, Message, Sender, Monitor, Call)

lgt_after_(Object, Message, Sender, Monitor, Call)
```

The first table stores *before* events while the second stores *after* events. These tables are accessed and updated by the event handling built-in predicates described in Chapter 6.

The last argument of each table entry, `Call`, contains the goal that will be called when a spied event occurs. This goal is constructed by the built-in predicate `define_events/5`. The goal functor is generated from the monitor identifier and the event handler method (`before/3` for *before* events and `after/3` for *after* events) in the same way as any other object predicate functor.

Assume, for example, then we want the object `spy` to be alerted every time the message `go/1` is exchanged between any two objects:

```
| ?- define_events(after, _, go(_), _, spy).
```

This call will generate the following table entry:

```
lgt_after_(Object, go(_), Sender, spy,
           spy0_after3(Object, go(_), Sender, spy, spy, spy)).
```

Note that the values of the *sender*, *this*, and *self* contextual information are all set to the monitor name. This is the expected value for *this* and *self*. However, there is no suitable value for *sender*, as the goal will be called directly by the message processing mechanism. Nevertheless, this raises no problems as the sender of the spied message is one of the arguments of the event handler method.

8.8 Limitations

This section describes two sets of limitations of the current Logtalk implementation: Prolog-related limitations and operating system-related limitations.

8.8.1 Prolog-related limitations

Most Prolog compilers implement different solutions for problems that are outside the scope of the current ISO Prolog standard. This results in limitations to the Logtalk implementation, which must remain compatible with most Prolog compilers.

Reloading entities

Interactive development of an application is characterized by frequent editing and reloading of source files. Ideally, reloading an entity would replace the old definition in memory. Since the Logtalk preprocessor compiles an entity source file to Prolog code, the resulting file will be compiled and loaded by calling the selected Prolog compiler. The ISO Prolog standard defines the syntax and semantics of compliant Prolog source code, but classifies the compilation and loading of programs as implementation-dependent features.

Fortunately, most compilers match the standard specification of compliant Prolog text. Mainly, all code is assumed static except for explicitly declared dynamic predicates. In addition, most compilers provide a reconsult-like built-in predicate that will replace old definitions when loading new clauses for an existing predicate. However, not all reconsult predicates behave as expected, particularly regarding dynamic predicates. For example, in some Prolog compilers, reloading a file containing directives for dynamic predicates does not remove from memory any existing clauses for those predicates.

Recompiling entities

Compiling and loading an application should only imply recompilation of source files that have been modified since the last time. This basic feature is implemented by accessing and comparing file modification dates. However, some Prolog compilers provide no support for retrieving file properties, making this feature compiler-dependent (controlled by a compiler option described earlier on this chapter) instead of a standard Logtalk feature.

Reporting syntax errors

The current version of the Logtalk preprocessor relies on the ISO Prolog specified predicate `read_term/3` for compiling an entity 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. In some Prolog compilers, this predicate reports useful information such as the source file line number where a syntax error occurred, but there is no agreed format for the error term. Since the predicate `read_term/3` can be used to read terms from sources other than files, this limitation results from our use of the predicate and not from a weakness of the ISO Prolog standard. A possible solution to this problem would be for Logtalk to implement its own parser instead of relying on the Prolog compiler.

Message sending operators

The message sending operators used by Logtalk, `::/1-2` and `^^/1`, although compatible with the set of predefined operators specified in the ISO Prolog standard, may conflict with the operator definitions of some Prolog compilers. For example, some compilers include constraint-solving extensions that use the operator `::/2` for declaring the domains of constraint variables. A plausible solution would be the definition, within the ISO standard, of hook operators and predicates for common Prolog extensions such as constraints and objects. As an example, note that the ISO standard already specifies a curly bracketed term notation (`{}/1`) and a predefined operator (`-->/2`) that can be used as hooks for implementing Definite Clause Grammars[4].

8.8.2 Operating system-related limitations

Some of the limitations of the current Logtalk version are a consequence of the goal of making our implementation portable across different operating systems. Trivial things

[4] In Logtalk, the curly bracketed notation is used as a control construct for bypassing the preprocessor during the compilation of entities.

such as valid file names, file path specifications, or end-of-lines in source files, differ from one operating system to another.

Parametric object file names

Sometimes, parametric objects have names whose corresponding file names are not valid in some operating systems. For example, consider the objects in the symbolic derivation and simplification example presented in Chapter 1. For the object `Op1*Op2`, the corresponding file name will be `*2.lgt`. This is an invalid file name in some operating systems that use the character "`*`" in regular expressions in order to match file names. Other similar examples will be parametric objects named `Op1/Op2` or `Op1:Op2`. The corresponding file names, respectively, `/2.lgt` and `:2.lgt`, are invalid in some operating systems that use the characters "`/`" and "`:`" as path separators.

A closely related issue is the compatibility of the file names with the regular expressions used in shell scripts (for example, the scripts included in the current implementation, which automate tasks such as converting XML documenting files to other formats).

These problems will eventually be solved when operating systems evolve to support less restrictive file naming.

Source file end-of-lines

Different operating systems use different characters for text file end-of-lines. When distributing source code, we need to convert the end-of-lines in the source files to comply with the target operating system; otherwise some Prolog compilers will fail to compile them, sometimes without any error message. Due to the use of full stops in Prolog as clause terminators, this was an unexpected problem. Fortunately, tools are available to easily batch convert whole directories of text files between different line endings.

8.9 Porting

The current Logtalk implementation comprises 320 predicates, amounting to 176 KB of Prolog code. Porting Logtalk to a specific Prolog compiler implies writing the corresponding configuration file. For many Prolog compilers, this was more problematic than anticipated. In the end, more time was spent on porting Logtalk than on writing the system itself. This section describes the results of trying to port Logtalk to thirty-seven versions of twenty-six Prolog compilers running on several flavors of Unix, Apple, and Microsoft operating systems.

8.9.1 Porting results

The most recent versions of eleven Prolog compilers broadly comply with the ISO Prolog standard. Four of these compilers are open source projects. Three are commercial products. Four other compilers are freely available, being three of them closed-source projects. Writing the configuration files for these compilers was an easy task to accomplish.

For two other Prolog compilers, the missing ISO Prolog predicates necessary for Logtalk were easy to replace. For the remaining compilers, the configuration files contain

workaround definitions for ISO Prolog predicates. These workarounds allow Logtalk to run some simple examples, but prevent its use for any serious work. One problem with the current preprocessor implementation of Logtalk is that, on occasion, Prolog compiler bugs have a negative impact on the perception of the quality of Logtalk itself, particularly if the compiler that the user is running is a commercial product.

In the end, working configuration files were written for thirty-one versions of twenty Prolog compilers. Six of the initial target compilers could not be supported due to their failure to implement or emulate basic ISO Prolog standard predicates. Four compiler versions need patches that cannot be applied using the configuration files. In addition, six compiler versions have buggy input/output predicates that imply editing of both Logtalk source files and Logtalk produced Prolog code files to workaround compatibility problems. Writing the configuration files for these compilers was a difficult and time-consuming task.

8.9.2 Porting reliability

Effective use of Logtalk in real-world problems implies evaluating the reliability of the selected Prolog port. For the current implementation, this depends on the quality of the corresponding configuration file and on the conformance of the selected Prolog compiler with the ISO standard. Thus, discussing port reliability is meaningful only to those Prolog compilers that comply broadly with the (now eight years old) standard.

Writing a quality configuration file depends on the quality of the documentation of the selected Prolog compiler. Quality documentation implies complete reference manuals, detailed release notes, and lists of known issues and bugs. Even when the documentation provides all necessary information, it is sometimes difficult to make any definitive statements about Logtalk compatibility for some of the supported compilers. Some configuration files are written using documentation available on-line, without any actual testing, due to lack of access to a copy of the corresponding Prolog compiler. Other configuration files are tested with only one of the operating systems supported by the Prolog compiler. Experience has shown that some bugs are platform-specific, and so, each Logtalk port would need to be tested in every supported operating system. Unfortunately, this is was not feasible for some compilers, in particular for commercial ones, due to lack of resources.

The ISO Prolog standard makes it possible to construct a suite of conformance tests against which a Prolog compiler could be evaluated [112, 113]. This is essential to ensure that a portable program, such as the current Logtalk implementation, runs as expected under a specific compiler, or that any differences in behavior are predictable. Unfortunately, judging from the number of bugs found during Logtalk development, testing compilers conformance is not a common practice among most Prolog vendors yet. The ideal solution would be the implementation of a conformance and certification process by an independent third-party entity. Until that happens, determining a port reliability would imply first certifying the standard conformance of the selected Prolog compiler, something that is out of the scope of this thesis work. Nevertheless, according to the users reports and to my own experience (that includes using Logtalk to teach object-oriented programming to undergraduate students), the reliability of the most commonly used ports is excellent.

8.9.3 Porting issues

The following notes detail some of the porting issues found while writing configuration files for compilers that deviate significantly from the ISO Prolog standard. These include some commercial systems that publicize their participation on the standardization process. The motivation for these ports was to allow any Prolog user to try out the Logtalk language using any available compiler.

Exception handling

Logtalk uses extensively the ISO Prolog predicates `catch/3` and `throw/1` for exception handling. Some compilers provide weak support, if any, for handling runtime exceptions. Definitions such as:

```
catch(Goal, _, _) :-
    call(Goal).

throw(Error) :-
    writeq(Error),
    abort.
```

allow some Logtalk examples to run, but are very limited workarounds. The nature of these predicates precludes functional user definitions for them.

A second problem will be whether we want to use exception handling only for our program-specific runtime errors or if we need to rely upon standard behavior for built-in predicates. For example, consider the predicate `arg/3`. In some Prolog compilers the following call:

```
| ?- arg(-1, Term, Arg).
```

will fail silently. In other compilers, the call will throw an exception. Unfortunately, this exception is not always the one specified by ISO Prolog standard for this case. If our code depends on `arg/3` throwing the correct exception in such erroneous cases, then we will have to write some workaround definition such as:

```
my_arg(N, Term, Arg) :-
    N < 0 ->
        throw(error(domain_error(not_less_than_zero, N)))
        ;
        arg(N, Term, Arg).
```

Applying this solution to all problematic built-ins, for all the possible exceptions, may result in a significant performance overhead to our programs.

A third problem was found on compilers that handle undefined predicates (that is, predicates with no clauses) as unknown predicates (ignoring dynamic, discontiguous, and multifile directives). If dummy definitions are not possible, this may force us to set the default action of the unknown predicate handler to fail for erroneous calls, losing a valuable aid in finding misspell predicate calls.

Input/output

Logtalk uses stream-based input/output predicates. Compilers that do not provide such predicates usually include the old Edinburgh set of input/output predicates. For those, a possible workaround will be to write definitions such as:

```
open(Stream, write, Stream) :-
    telling(Current),
    tell(Stream),
    tell(Current).

read_term(Stream, Term, _) :-
    seeing(Current),
    see(Stream),
    read(Term),
    see(Current).
```

With a little more work, we can also implement some necessary read and write options. For example:

```
write_term(Stream, Term, Options) :-
    telling(Current),
    tell(Stream),
    (member(quoted(true), Options) ->
        writeq(Term)
        ;
        write(Term)),
    tell(Current).
```

However, these are very incomplete and fragile workarounds.

Operators

The current Logtalk implementation assumes that its declared operators remain active (once the Logtalk preprocessor and runtime files are loaded) until the end of the Prolog session. This is the usual behavior of most Prolog compilers. However, operators are a common source of portability problems.

Predefined operators Some compilers declare atoms such as `public`, `dynamic`, or `mode` as operators. To avoid syntax errors, all occurrences of these atoms are written between parenthesis in the current Logtalk implementation. However, parsing errors may also result from differences in precedence and/or type of common operators. Most compilers allow the redefinition of built-in operators, but this may clash with other programs that the user may want to run concurrently.

Operators and input/output predicates One particularly nasty problem found in some compilers is that predicates such as `writeq/1` writes code that the built-in `read/1` predicate fails to parse correctly. While sometimes we get a parsing error, in other cases the code that is read back is not the same as that we have written. One instance of this

problem, which was found in four Prolog compilers (both commercial and academic), happens when writing a term that uses the negation operator. Goals such as:

```
writeq(\+ (a, b, c))
```

may output:

```
\+(a, b, c)
```

omitting the necessary whitespace character between the operator and the parenthesis. Because `\+` is a valid functor and there is no `\+/3` predicate, this may create elusive bugs in our programs. Sometimes the above goal outputs correct code, but a call such as:

```
write(\+ \+ (a, b, c))
```

may incorrectly output:

```
\+ \+(a, b, c)
```

The double negation is a construct commonly used to check if a goal is true without instantiating any free variables. The only (cumbersome) workaround may be to test every term we write and, if needed, write the operator first, then a space, and finally the operand.

Minimizing the use of operators The ISO Prolog standard defines a predicate named `write_canonical/1` that is intended for writing code that must be read back. This predicate outputs terms without using operator notation and quoting them if necessary. Logtalk uses this predicate for writing the Prolog code files that result from the compilation of Logtalk source files. However, until recently, some Prolog compilers provided no implementation for this predicate.

Theoretically, the output from any Prolog compiler produced with the predicate `write_canonical/1` should be the same. In practice, this does not always happen. For example, a goal such as:

```
..., write_canonical(dynamic), ...
```

will output the atom `dynamic` between parenthesis in some compilers (usually, because this atom is declared as an operator) but not in others. As consequence, depending on the used Prolog compiler, the current Logtalk implementation may generate Prolog code files that will not be portable across all Prolog compilers. In addition, some compilers have buggy implementations of the predicate `write_canonical/1` where the code that is written cannot be parsed by the compilers own predicates for compiling and loading Prolog code.

Access to the operating system

Predicates for retrieving system time, system date, and timing information are available on most Prolog compilers. In respect to file system access, most Prolog compilers define predicates for testing if a file exists and for changing the current working directory. Some commercial systems (and also a few academic ones) provide very good operating system interfaces. However, the limited operating system access of most Prolog compilers prevents the widespread implementation of useful development features, as already discussed on this chapter.

Built-in predicates

When selecting the set of built-in predicates, shared by a set of compilers, in order to write a portable program, two kinds of problems may arise:

1. Equivalent built-ins with different names.
2. Built-in predicates with the same name but different call modes, different argument types, or different behavior in case of invalid calls.

Defining our own predicate to call the equivalent built-in predicate in each Prolog compiler easily solves the first problem. In Logtalk, an example of this is the predicate used to check if a file exists in the current working directory.

The second problem may also be solved by defining our own predicate. This may result in performance penalties, depending on the complexity of the workaround definition and on how often it is called by our program. Some compilers allow the redefinition of built-in predicates. This may provide a better solution if the problem only occurs with some of the target compilers. However, we must be aware that this may cause incompatibilities with other programs (or even with the compiler itself) that may rely on the replaced definitions. A common example is the built-in predicate `predicate_property/2`, which is provided by most Prolog compilers for retrieving predicate properties. The problem is that different compilers may return different atoms to represent the same properties. For instance, for a static predicate, some compilers return the atom `static`, while others return the atom `compiled`. The same happens for dynamic predicates: some compilers return the atom `dynamic`, while others return the atom `interpreted`.

Database updates

The ISO standard defines a *logical view* for database updates. In this view, the database is frozen while a goal is executed. Any database update will only affect the next goals. However, some Prolog compilers still use *immediate update* semantics where any assert or retract call may affect the goal under execution. For programs that make use of dynamic code, compilers implementing different database update semantics may lead to different results for some goals. This should not affect the porting of the current Logtalk implementation *per se* but may affect the portability of Logtalk programs. Careful use of database update predicates as explained, for example, in [114], can help to avoid these potential problems.

Documentation and developer support

Some of the problems described above could be minimized with better compiler documentation, including a summary of the differences between a Prolog implementation and the ISO standard. Lists of known issues and bugs are essential for avoiding long debugging sessions caused not by bugs in our code but by compiler-related problems. In the development of Logtalk, I have contributed to uncover several problems, both in commercial and academic Prolog compilers. It is interesting to note that, while academic systems are patched within a few days, sometimes within a few hours (with gratitude for the bug reports from its authors), a few support teams of commercial systems either refuse to acknowledge the problems ("it's not a bug, it's a feature"; not documented of course, but still a feature!) or had harsh reactions to bug reports (including stop replying email messages).

8.10 Summary

This chapter described solutions for implementing several object-oriented features not found in other Prolog object-oriented extensions. These include: public, protected, and private predicates; public, protected, and private inheritance; separation of interface from implementation using protocols; code reuse through categories; and support for both prototypes and classes in the same application.

The current implementation fully implements all the Logtalk language features as described and specified on this thesis. In addition, the technical goal of ensuring compatibility with most Prolog compilers has been achieved. However, the reliability of some ports is weak due to the lack of standard conformance, good documentation, and access to Prolog compiler copies for testing.

A substantial number of Prolog compilers and compiler versions currently supported by Logtalk are no longer maintained by its authors. Furthermore, some commercial compilers still do not support basic ISO Prolog standard functionality such as exception handling and stream-based input/output predicates. Future Logtalk versions will no longer support those Prolog compilers. This will make porting and maintenance much easier. It will also allow adding new features (that need Prolog support only available in recent compilers) to the Logtalk compiler.

Most limitations of the current implementation could be solved by a closer integration with a suitable Prolog compiler. A good candidate would be a standard-compliant compiler, available under an open source license, compatible with most operating systems, with a good operating system interface, and with either no module system or with a module system implementation that could be easily removed. Preliminary experiments with GNU Prolog [111] have shown the feasibility of this solution. Thus, a complete implementation that would not need any supporting Prolog compiler could also be provided. However, this approach would not work for Logtalk users who use libraries and features only available on specific Prolog systems. Those users would be limited to use the preprocessor version of Logtalk.

Conclusions

This chapter presents the most relevant contributions of this thesis, some supplementary considerations on Logtalk support for reflection and on using Logtalk in the classroom, as well as the planned future development of the Logtalk language. Complete and more detailed conclusions about each language feature of are provided in the end of each chapter.

Logtalk can be described as a multi-paradigm language that supports logic programming, object-oriented programming, and event-driven programming. However, Logtalk goal was not to only *support* these programming paradigms but to *integrate* them. The integration was made by, first, reinterpreting object concepts in the context of logic programming and, second, by reinterpreting event concepts in the context of object-oriented programming.

This chapter begins by evaluating Logtalk as a Prolog object-oriented extension and as an interpreted, interactive object-oriented programming language. Secondly, the relevance of event-driven programming in the context of object-oriented languages is described. Thirdly, category-based composition, and its support for component-based programming, is summarized. Fourthly, Logtalk native support for reflection is examined. Then, the Logtalk support for automatic program documentation is presented. Next, the experience of using Logtalk in the classroom is described, followed by some data on the Logtalk distribution numbers. Finally, the roadmap for future development is presented.

Logtalk as a Prolog object-oriented extension

Logtalk reinterprets the concept of object as a set of predicate directives (declarations) and clauses (definitions). Thus, message sending is reinterpreted as proof construction using the predicates defined for the receiving object. Inheritance mechanisms allow us to define the *complete database* of an object. A method is then simply the predicate definition selected from an object complete database in order to answer a message. By reinterpreting the concepts of object, message, and method in logic programming terms, a simple mapping is established between Logtalk semantics and the familiar Prolog semantics.

In general terms, this reinterpretation of object concepts is shared by most Prolog object-oriented extensions. To evaluate and compare Logtalk with other object-oriented extensions and Prolog module systems, the following criteria will be used: compatibility with Prolog compilers, language syntax, interpretation of the concept of object, feature set, and working environment.

Logtalk compatibility

Logtalk is the only Prolog object-oriented extension available today that has been designed from scratch for compatibility with most compilers and with the ISO Prolog standard. This design goal sets it apart from other Prolog extensions. The preprocessor solution adopted for the Logtalk implementation allows it to run on most computers and operating systems for which a modern Prolog compiler is available. Specifically, the current Logtalk version is compatible with thirty-one versions of twenty Prolog compilers.

Logtalk syntax

Logtalk uses, whenever possible, standard Prolog syntax, and defines elegant language constructs, in line with the current practice and expectations of Prolog programmers. This helps to smooth the learning curve for Prolog programmers. This is more than a syntactic sugar issue. For example, Logtalk enables existing Prolog code to be encapsulated in objects without any changes. Only when a predicate needs to call other object predicates, would minimal changes be required. Thus, easy conversion of old Prolog programs is ensured.

The role of objects in logic programming

The primary purpose of objects in Logtalk is the encapsulation and reusing of code, thus decoupling this functionality from the theoretical issues of dynamic state change in logic programming. As such, Logtalk provides a practical view, rather than a theoretical view, of the role of objects in logic programming in general, and in Prolog in particular. By focusing on the encapsulation and code reuse proprieties of objects, Logtalk aims to be an effective tool for solving software engineering problems in Prolog programming.

Implementation solutions for object-oriented concepts

Logtalk shows how to implement the main object-oriented concepts in Prolog. These include concepts not found individually on most Prolog object-oriented extensions such as: support for both classes and prototypes; metaclasses; protocols and protocol hierarchies; public, protected, and private predicates; and public, protected, and private inheritance. Thus, Logtalk is likely one of the most complete Prolog object-oriented extension available today. In addition, Logtalk shows how to implement other important concepts that are not available on other object-oriented languages, such as categories and event-driven programming. Unlike other object-oriented extensions that are either proprietary or depend heavily on the specifics of the native module systems, Logtalk implementation solutions are fully compatible with any compiler that complies with the Part I of the ISO Prolog standard.

Objects as a replacement for modules

Logtalk objects provide an alternative to the use of Prolog modules, either as implemented in most Prolog compilers or as specified in the Part II of the ISO Prolog standard. Like Prolog modules, Logtalk prototypes can be defined as stand-alone encapsulation entities. In addition, although Logtalk does not provide a direct replacement for module import and export directives, the extension relation between prototypes, together with

protocol implementation and category importation relations, allows equivalent functionality. Logtalk objects have several important advantages over Prolog modules:

- Logtalk predicate scope directives ensure data hiding, a missing feature in the ISO standard for Prolog modules.

Logtalk message sending mechanism, built-in methods, and built-in predicates enforce predicate scope directives. The ISO standard specifies that any module predicate can be called using explicit module qualification; it considers that mechanisms to enforce data hiding are implementation-dependent features.

- Separation between interface and implementation.

Unlike module interfaces, Logtalk protocols can be implemented by any number of objects. Conversely, an object may implement several protocols.

- Compatibility with existing Prolog compilers.

Logtalk is compatible with almost all modern Prolog compilers. The ISO standard for Prolog modules is still to being adopted by most Prolog vendors, in part because of differences with existing and widely used module systems.

- The ISO standard specifies two incompatible ways of declaring metapredicates. No such nuisances exist in Logtalk.

Some things cannot be specified within a standard and must be considered as implementation-dependent features. The syntax for declaring metapredicates is clearly not one of them.

- Logtalk provides a number of valuable features in the development of large-scale projects, which are outside the scope of module systems.

These include predicate reuse and specialization through inheritance and composition, event-driven programming, reflection, and automatic generation of documentation.

Working environment and other practical matters

Logtalk working environment is limited to the subset of common features of the compatible Prolog compilers. For example, there is no common set of predicates for operating system access (so restricting the functionality of the Logtalk compiler) or a common standard for constructing graphical user interfaces. Proprietary object-oriented extensions, developed to work with a single Prolog compiler, are able to provide a richer working environment, taking advantage of unique features of a compiler and its hosting operating system. Nevertheless, within its compatibility restrictions, Logtalk tries to provide a learning and working environment similar to other Prolog compilers and object-oriented extensions that use text-based development tools. For editing source files, the current Logtalk distribution includes syntax-coloring configuration files for popular text editors, along with text templates for defining new entities and entity predicates. This improves the programming experience, particularly by helping to avoid syntax errors when writing entity and predicate directives. Also included are a number of programming examples and extensive documentation, which comprises a user manual, a reference manual, and programming tutorials.

Outside the academic world, these practical matters are as important as the technical features and scientific achievements of the language itself. As such, they are fundamental in building a user community, who will use the Logtalk language as a tool to solve real problems.

Logtalk as an object-oriented programming language

Logtalk extends Prolog, in the same way as CLOS extends LISP or Objective-C extends C. As a programming language in its own right, Logtalk shares features with common object-oriented languages. However, there are also some important differences because of its Prolog roots. The most significant one is that Logtalk eliminates some dichotomies deeply established on most object-oriented languages. These dichotomies are often used as a way of characterizing and classifying object-oriented languages. Specifically, Logtalk makes no distinction between variables and methods, allows most language elements to be either dynamic or static, and integrates classes and prototypes in the same language. Despite the first two points are common to some Prolog object-oriented extensions, they are worth summarizing here. Unlike Logtalk, almost all object-oriented languages are strictly defined as either class-based or prototype-based languages. Most of them are characterized by a clear distinction between variables and methods, what is static and what is dynamic, and what must be achieved at compile time or can be performed at runtime.

Predicates as both variables and methods

Logtalk object predicates unify the concepts of object methods and object variables, therefore simplifying the language semantics. Predicates remove the dichotomy between state and behavior: a predicate simply states what is true about an object. Predicates may be used to implement both variables and methods, but such a distinction is always optional. It follows that we no longer need separate definition, inheritance, and scope rules for state and behavior. In addition, both state and behavior can be easily shared along inheritance links or defined locally. This allows us, for example, to define methods in instances and to share object state easily among descendant objects without the need of first formalizing concepts such as shared instance variables.

Static and dynamic language elements

Logtalk objects, protocols, categories, and predicates can be either dynamic or static. Predicates may be asserted into, and abolished from both static and dynamic objects at runtime. Objects, protocols, and categories can be defined either in a source file or created dynamically at runtime. If contained in a source file, they can be defined as either static or dynamic entities. As such, we are not constrained, for example, to define classes as static entities and instances as runtime-only objects. We may define an instance in a source file in the same way as a class may be defined. This is consistent with Logtalk primary view of objects as encapsulation units.

Support for both prototypes and classes

Logtalk is a neutral, unbiased language, supporting both prototype-based and class-based programming. We can use both types of objects at the same time and freely

exchange messages between them. Logtalk classes, instances, and prototypes are simply objects — encapsulation entities — characterized by different rulesets for accessing their own predicates and for reusing predicates inherited from other objects. Classes and prototypes share the same built-in predicates for creating, abolishing, and enumerating objects. In addition, they share the same built-in methods for dynamic object modification and the message sending mechanisms. Moreover, protocols can be implemented, and categories can be imported, by prototypes, classes, and instances.

As a neutral language, Logtalk enables the definition and coexistence in the same application of several types of object systems, all of them supported in an equal basis. Class-based designs may define a single object hierarchy, as in Smalltalk and Java, or multiple, independent hierarchies as in C++. In addition, one may choose to implement reflective systems through metaclasses or simply use classes as instance factories. Multiple hierarchies of prototypes are also supported, of course. Both class and prototype hierarchies may use only single inheritance or take advantage of multi-inheritance support. This flexibility is an important asset when using Logtalk for teaching object-oriented programming.

Event-driven programming

Logtalk provides a conceptual integration of event-driven programming into the object-oriented programming paradigm. The key for this integration is the interpretation of message sending as the only event that occurs in an object-oriented program. Thus, Logtalk reinterprets the concepts of event, monitor, event notification, and event handler in terms of objects, messages, and methods. This allows us to stay within the object-oriented programming paradigm when writing event-driven programming code.

Two important results emerge from Logtalk programming practice in using events to implement complex dependency relations between objects. These results are not specific to Logtalk. Instead, they apply to most object-oriented programming languages. First, event-driven programming is an essential feature of object-oriented languages for achieving a high level of object cohesion and avoiding unnecessary object coupling on applications where object relations imply constraints on the state of participating objects. Dependency mechanisms, as found in Smalltalk and in other languages, provide only a partial solution, which can only be used when object methods contain — or can be modified to contain — calls to the dependency mechanism methods. Second, events and monitors must be supported as language primitives, integrated with the message sending mechanisms. This is an essential requirement from a performance point of view, which precludes the implementation of event-driven programming at the application layer or through language libraries. Native language support is fundamental in making event-driven programming an effective tool for problem solving.

Category-based composition

Categories are the basis of component-based programming in Logtalk. Categories provide finer, functionaly-cohesive units of code that can be imported by any object. Thus, categories play a role dual to that one played by protocols for interface encapsulation. In addition, categories provide several development benefits such as incremental compilation, updating an object — without recompiling it — by updating its imported

categories, and refactoring of complex objects into more manageable and reusable parts.

Category-based composition, inheritance, and instance variable-based composition provide complementary forms of code reuse. Categories implement a composition mechanism where a category interface becomes part of the interface of an object importing it. This is unlike instance variable-based composition, but similar to what happens with inheritance. Logtalk support for public, protected, and private category importation provides further flexibility. By applying the principles of *separation of concerns* common to component-based programming approaches, categories provide alternative solutions to multi-inheritance designs that can be applied in the context of single inheritance languages.

The concept of category has no dependencies on Logtalk-specific features. Categories are compiled using similar techniques to those applied to the compilation of objects and object hierarchies, with some features requiring the use of dynamic binding. As such, categories can be implemented in other object-oriented languages in order to add support for component-based programming.

Reflection

Logtalk inherits Prolog metaprogramming features, which are a form of reflection. In addition, Logtalk provides a framework for reflective computations supporting both structural and behavioral reflection. While reflection is usually provided by *reification* of language constructs, Logtalk supports reflection through built-in language features such as built-in predicates, built-in methods, and runtime mechanisms, which are not written in Logtalk itself. This implies that some Logtalk features, such as the message sending mechanism or the inheritance algorithms, are fixed and cannot be tailored by the programmer. On the other hand, this enables Logtalk reflection support to be fully optimized for runtime performance.

Logtalk supports structural reflection through a set of built-in methods for enumerating object predicates and their properties, and a set of built-in predicates for enumerating entities (objects, protocols, and categories), their properties, and the relations between them. All reflection built-in predicates and built-in methods can be applied to prototypes, instances, and classes.

Logtalk supports behavioral reflection through event-driven programming. Therefore, behavioral reflection is restricted to computations about the messages exchanged between objects. Nevertheless, events enable easy construction of common reflective applications such as code profilers and debuggers.

Logtalk also supports reflection through the definition of metaclasses. Any class may have its own metaclass as in Smalltalk or share its metaclass with other classes. Metaclasses may be defined for all classes or just for some of them. Thus, any class-based design with metaclasses is supported.

Program documentation

Logtalk emphasis on the documentation of programs has its roots on literate programming concepts. Logtalk support for program documentation differs from most programming languages in four key points:

- Documenting files are automatically generated whenever an entity is compiled.

Logtalk uses a single tool — its compiler — to both compile code and extract documentation. By writing some simple scripts (examples of which come with the current implementation), collecting and processing documenting files can be fully automated.

- The documenting files are XML compliant files. These files contain all information on an entity that might have relevant documentation meaning, including entity relations and entity predicates.

The XML format allows us to represent documenting information without concerning ourselves on how it will be used. XML documenting files can be easily converted to any human-readable format such as PDF (for printing) or HTML (for on-line reading). In addition, XML documenting files may also be parsed for other useful purposes such as collecting program metrics.

- The structure of the documenting files is an integral part of the language specification, along with documenting directives for entities and entity predicates.

Thus, in Logtalk, program documentation is viewed as a fundamental part of the language specification.

- All documenting information is expressed in the Logtalk language itself. There is no additional, specific documenting language that needs to be mastered before programs can be documented.

Logtalk documenting directives are user-extensible. This allows programmers to easily express documenting information that cannot be deduced from the program itself. This differs from typical literate programming solutions, which use specially formatted comments for documenting programs.

Logtalk in the classroom

Logtalk is being used at UBI (University of Beira —nterior) to teach object-oriented programming and object-oriented extensions to logic programming to undergraduate students. Teaching object-oriented concepts using Logtalk has provided interesting results. Common object-oriented languages such as C++ and Java are class-based. These languages inherit syntax and concepts from imperative languages such as C. They require understanding concepts such as static versus dynamic allocation, method argument and return types, library imports, and main methods, in order to write simple programming examples. These concepts get in the way of teaching key object-oriented concepts like encapsulation, message sending, or inheritance. In contrast, Logtalk objects encapsulate predicates. There is no need to talk about methods and variables, or function and data members, before defining a simple predicate and sending the corresponding message. In addition, the Logtalk support for both prototypes and classes means that we can teach basic object-oriented concepts using simpler prototype hierarchies before explaining the difference between classes and instances or between instantiation and specialization mechanisms. There are no `main`, `static`, `void`, `include` or `import` keywords cluttering and distracting the student from the concept that a given example tries to convey, as it happens in C++ or Java. There is no rich, integrated, development environment whose basics need to be understood and mastered before simple programs can be written as in Smalltalk. A simple text editor suffices. For students

with a basic knowledge of Prolog programming, Logtalk is an ideal learning tool for a smooth transition from logic programming to object-oriented programming, due to the use of familiar Prolog syntax and semantics and to the support of a wide range of object-oriented systems.

Logtalk in numbers

In the last two years, Logtalk was downloaded an average of 270 copies per month[5] (9 copies per day). In addition, Logtalk is distributed with YAP, an open-source Prolog compiler. New releases are in general announced only in the Logtalk mailing list (that is subscribed by around 70 users) and in the Freshmeat web site [115] (a web index of cross-platform software, mostly open-source products; around 10 users have subscribed to new Logtalk releases through this web site). The Logtalk web site is linked from around one hundred web sites, ranging from links in personal pages to web directories of programming resources. These are modest numbers when compared to the estimated size of the Prolog user community. They show that there is enough interest on Prolog object-oriented extensions to continue Logtalk development, but also that more efforts need to be done to increase the use of Logtalk.

Future work

Short-term development plans include making the language and more appealing to potential users by improving its documentation, library, and examples. A key goal is to continue to make Logtalk an attractive language for teaching object-oriented concepts to undergraduate students. In addition, complex examples must be written to illustrate the potential of Logtalk to deal with software engineering problems.

Mid-term plans include publishing this thesis results and improving the Logtalk compiler technology. The work spent on implementing Logtalk, and on the writing of this thesis, must be translated into technical and scientific publications, which will describe major features such as component-based programming through categories, the support for both prototypes and classes, and the integration of event-driven programming and object-oriented programming. In respect to the compiler implementation, several improvements are planned:

- Support for outdated versions of Prolog compilers will be dropped. This will ease Logtalk maintenance. In addition, this will allow us to take advantage of features only available in recent Prolog compilers.
- Message sending performance will be optimized by using static binding whenever possible.
- A new implementation will be developed by merging the Logtalk code base with an open-source Prolog compiler, resulting in a stand-alone Logtalk version. This will be an alternative implementation, complementing, but not replacing, the current preprocessor implementation of Logtalk.
- A suite of tests will be developed to help certify Prolog compatibility with Logtalk.

[5] http://www.logtalk.org/statistics.html

A long-term goal is to make Logtalk a *de facto* standard for object-oriented programming in Prolog. One of the major hurdles to overcome is to fight the perception of Prolog modules as a "good enough" solution for software engineering problems in Prolog applications.

Appendix A

Logtalk Grammar

This appendix formalizes the syntax of the Logtalk language through a grammar. The Logtalk grammar syntax used here, is a simplification of the Backus-Naur Form notation [116, 117]. Non-terminal symbols are represented in *italics*. Non-terminal symbols common to Prolog (such as *atom*) are not defined here. Their definition can be found in the ISO Prolog Standard [2]. Terminal symbols are represented using a `fixed width font`. Alternative constructs are separated by a vertical bar. Optional constructs are enclosed in "[ ]". The grammar productions are listed in a top-down fashion, starting with the entity enclosing directives. Most productions are annotated with brief remarks. For more details and examples, please see the next appendix.

A.1 Entity types

There are three entity types in Logtalk: objects, categories, and protocols. They are the smallest unit of code that can be compiled by Logtalk.

entity →
    object |
    category |
    protocol

A.2 Entity definitions

Every entity has a textual representation and can thus be defined in a source file. The textual representation begins with a starting directive, followed by the entity directives and clauses, and ends with an ending directive.

A.2.1 Object definition

object →
    begin_object_directive [*object_directives*] [*clauses*] *end_object_directive*

begin_object_directive →
    `:- object(` *object_identifier* [`,` *object_relations*] `).`

end_object_directive →
    `:- end_object.`

object_relations →
    prototype_relations |
    non_prototype_relations

prototype_relations →
    prototype_relation |
    prototype_relation `,` *prototype_relations*

prototype_relation →
    implements_protocols |
    imports_categories |
    extends_objects

non_prototype_relations →
    non_prototype_relation |
    non_prototype_relation `,` *non_prototype_relations*

non_prototype_relation →
    implements_protocols |
    imports_categories |
    instantiates_classes |
    specializes_classes

Note that the object relations can be either prototype relations or class/instance relations. In addition, the ordering of the object relations above is not mandatory. It is, however, the recommended ordering because it reflects the search strategy used by Logtalk when looking up predicate declarations and definitions.

A.2.2 Category definition

category →
    begin_category_directive [*category_directives*] [*clauses*] *end_category_directive*

begin_category_directive →
    `:- category(` *category_identifier* [`,` *implements_protocols*] `).`

end_category_directive →
    `:- end_category.`

A.2.3 Protocol definition

protocol →
    begin_protocol_directive [*protocol_directives*] *end_protocol_directive*

begin_protocol_directive →
    `:- protocol(` *protocol_identifier* [`,` *extends_protocols*] `).`

end_protocol_directive →
    `:- end_protocol.`

A.3 Entity relations

Entity relations describe all possible relations between objects, protocols, and categories.

A.3.1 Implemented protocols

An object or category may implement a number of protocols. By default, the implementation is public. To restrict the scope of the implemented predicates, the implemented protocol identifier may be preceded by a scope keyword.

implements_protocols →
    `implements(` *implemented_protocols* `)`

implemented_protocols →
    implemented_protocol |
    implemented_protocol_sequence |
    implemented_protocol_list

implemented_protocol →
    protocol_identifier |
    scope `::` *protocol_identifier*

implemented_protocol_sequence →
    implemented_protocol |
    implemented_protocol `,` *implemented_protocol_sequence*

implemented_protocol_list →
    `[` *implemented_protocol_sequence* `]`

A.3.2 Extended protocols

A protocol may extend a number of other protocols. By default, the extension is public. To restrict the scope of the predicates in an extended protocol, the extended protocol identifier may be preceded by a scope keyword.

extends_protocols →
    `extends(` *extended_protocols* `)`

extended_protocols →
    extended_protocol |
    extended_protocol_sequence |
    extended_protocol_list

extended_protocol →
    protocol_identifier |
    scope `::` *protocol_identifier*

extended_protocol_sequence →
    extended_protocol |
    extended_protocol `,` *extended_protocol_sequence*

extended_protocol_list →
    `[` *extended_protocol_sequence* `]`

A.3.3 Imported categories

An object may import a number of categories. By default, the importation is public. To restrict the scope of the imported predicates, the category identifier may be preceded by a scope keyword.

imports_categories →
    `imports(` *imported_categories* `)`

imported_categories →
    imported_category |
    imported_category_sequence |
    imported_category_list

imported_category →
    category_identifier |
    scope `::` *category_identifier*

imported_category_sequence →
    imported_category |
    imported_category `,` *imported_category_sequence*

imported_category_list →
    `[` *imported_category_sequence* `]`

A.3.4 Extended objects

An object may extend a number of objects, resulting in a prototype hierarchy. By default, the extension is public. To restrict the scope of the predicates inherited from an extended object, the extended object identifier may be preceded by a scope keyword.

extends_objects →
    `extends(` *extended_objects* `)`

extended_objects →
    extended_object |
    extended_object_sequence |
    extended_object_list

extended_object →
    object_identifier |
    scope `::` *object_identifier*

extended_object_sequence →
    extended_object |
    extended_object `,` *extended_object_sequence*

extended_object_list →
    `[` *extended_object_sequence* `]`

A.3.5 Instantiated objects

An object (an instance) may instantiate a number of objects (its classes). By default, the instantiation is public. To restrict the scope of the predicates declared in an instantiated class, the class identifier may be preceded by a scope keyword.

instantiates_classes →
    `instantiates(` *instantiated_objects* `)`

instantiated_objects →
    instantiated_object |
    instantiated_object_sequence |
    instantiated_object_list

instantiated_object →
    object_identifier |
    scope `::` *object_identifier*

instantiated_object_sequence →
    instantiated_object |
    instantiated_object , *instantiated_object_sequence*

instantiated_object_list →
    [*instantiated_object_sequence*]

A.3.6 Specialized objects

An object (a class) may specialize a number of objects (its superclasses), resulting in a class-based or generalization-specialization hierarchy. By default, the specialization is public. To restrict the scope of the predicates inherited from a specialized class, the specialized class identifier may be preceded by a scope keyword.

specializes_classes →
    `specializes(` *specialized_objects*)

specialized_objects →
    specialized_object |
    specialized_object_sequence |
    specialized_object_list

specialized_object →
    object_identifier |
    scope :: *object_identifier*

specialized_object_sequence →
    specialized_object |
    specialized_object , *specialized_object_sequence*

specialized_object_list →
    [*specialized_object_sequence*]

A.3.7 Entity relation scope

The relation between two entities can be restricted by using one of the following scope keywords:

scope →
    `public` |
    `protected` |
    `private`

Note that when an entity scope keyword is absent, the entity relation is public by default. This is the most common case, allowing us to simplify our code by removing redundant keywords.

A.4 Entity identifiers

It is important to note that all entity identifiers share the same namespace. That is to say, that we cannot have two entities of different types with the same identifier.

entity_identifiers →
    entity_identifier |
    entity_identifier_sequence |
    entity_identifier_list

entity_identifier →
    object_identifier |
    protocol_identifier |
    category_identifier

entity_identifier_sequence →
    entity_identifier |
    entity_identifier , *entity_identifier_sequence*

entity_identifier_list →
    [*entity_identifier_sequence*]

A.4.1 Object identifiers

An object identifier is either an atom or a compound term (in case of parametric objects).

object_identifiers →
    object_identifier |
    object_identifier_sequence |
    object_identifier_list

object_identifier →
    atom |
    compound

object_identifier_sequence →
    object_identifier |
    object_identifier , *object_identifier_sequence*

object_identifier_list →
    `[` *object_identifier_sequence* `]`

A.4.2 Category identifiers

The idea behind categories is to encapsulate functionally cohesive sets of predicates. The category identifier should reflect this functionality. For example, if a category contains predicates related to event monitoring, then "`monitoring`" is an appropriated category identifier while "`monitor`" will be a poor choice because a category is not an object as the name suggests.

category_identifiers →
    category_identifier |
    category_identifier_sequence |
    category_identifier_list

category_identifier →
    atom

category_identifier_sequence →
    category_identifier |
    category_identifier `,` *category_identifier_sequence*

category_identifier_list →
    `[` *category_identifier_sequence* `]`

A.4.3 Protocol identifiers

We often want to name a protocol using an identifier similar to the object that implements it. By convention, this is done by appending the letter "`p`" to the object identifier. For example, if we have a "`list`" object, we may encapsulate the predicate declarations in a protocol named "`listp`". The same convention applies to the definition of a category protocol.

protocol_identifiers →
    protocol_identifier |
    protocol_identifier_sequence |
    protocol_identifier_list

protocol_identifier →
    atom

protocol_identifier_sequence →
    protocol_identifier |
    protocol_identifier `,` *protocol_identifier_sequence*

protocol_identifier_list →
    `[` *protocol_identifier_sequence* `]`

A.5 Directives

Logtalk defines two sets of directives: entity directives and predicate directives. Entity directives apply to an entity as a whole. Predicate directives only affect the compilation of the specified predicates.

A.5.1 Entity directives

For details about the syntax of the "`info/1`" directive, listed below, and available for all entity types, please see the next section on predicate directives.

Object directives

object_directives →
    object_directive |
    object_directive object_directives

object_directive →
    `:- initialization(` *callable* `).` |
    `:- dynamic.` |
    `:- uses(` *object_identifiers* `).` |
    `:- calls(` *protocol_identifiers* `).` |
    `:- info(` *info_list* `).` |
    predicate_directives

Protocol directives

protocol_directives →
    protocol_directive |
    protocol_directive protocol_directives

protocol_directive →
    `:- initialization(` *callable* `).` |
    `:- dynamic.` |
    `:- info(` *info_list* `).` |
    predicate_directives

Category directives

category_directives →
    category_directive |
    category_directive category_directives

category_directive →
    `:- initialization(` *callable* `).` |
    `:- dynamic.` |
    `:- uses(` *object_identifiers* `).` |
    `:- calls(` *protocol_identifiers* `).` |
    `:- info(` *info_list* `).` |
    predicate_directives

A.5.2 Predicate directives

predicate_directives →
    predicate_directive |
    predicate_directive predicate_directives

predicate_directive →
    scope_directive |
    mode_directive |
    metapredicate_directive |
    info_directive |
    operator_directive |
    dynamic_directive |
    discontiguous_directive

The "`operator`", "`dynamic`", and "`discontiguous`" directives are not described here. They follow the definitions found in ISO Prolog Standard, with the one difference that their scope is the containing object (or category) instead of the whole Prolog database.

Scope directive

This is the most important predicate directive. Without it, a predicate is local to an object (or category) and cannot be called from other objects or seen by the predefined reflection methods. That is not to say that we must have a scope directive for each predicate. If we use many auxiliary predicates, then we may choose to avoid writing a scope directive for each one of them.

scope_directive →
    `:- public(` *predicate_indicator_term* `).` |
    `:- protected(` *predicate_indicator_term* `).` |
    `:- private(` *predicate_indicator_term* `).`

predicate_indicator_term →
    predicate_indicator |
    predicate_indicator_sequence |
    predicate_indicator_list

predicate_indicator_sequence →
    predicate_indicator |
    predicate_indicator `,` *predicate_indicator_sequence*

predicate_indicator_list →
    `[` *predicate_indicator_sequence* `]`

Mode directive

The mode directive is used to declare predicate call templates, including the number of solutions or proofs associated with each template. It can also be used to declare that a call template will result in a runtime error. The argument's type can be appended to its instantiation mode.

mode_directive →
    `:- mode(` *predicate_mode_term* `,` *number_of_solutions* `).`

predicate_mode_term →
    atom `(` *mode_terms* `)`

mode_terms →
    mode_term |
    mode_term `,` *mode_terms*

mode_term →
    `@` [*type*] |
    `+` [*type*] |
    `-` [*type*] |
    `?` [*type*]

type →
    logtalk_type |
    prolog_type |
    user_defined_type

logtalk_type →
    `object` | `category` | `protocol` |
    `event`

prolog_type →
    `term` |
    `nonvar` | `var` |
    `compound` |
    `atomic` | `atom` |
    `number` | `integer` | `float`

user_defined_type →
    atom |
    compound

number_of_solutions →
    `zero` | `zero_or_one` | `zero_or_more` |
    `one` | `one_or_more` |
    `error`

Metapredicate directive

Metapredicate directives are mandatory for every user-defined metapredicate in order to properly compile the predicate definition.

metapredicate_directive →
    `:- metapredicate(` *metapredicate_mode_indicator* `).`

metapredicate_mode_indicator →
    atom `(` *metapredicate_terms* `)`

metapredicate_terms →
    metapredicate_term |
    metapredicate_term `,` *metapredicate_terms*

metapredicate_term →
    `::` | `*`

The "`::`" symbol indicates that the corresponding argument will be called as a goal in the predicate definition. The "`*`" symbol is used for normal, non-meta arguments.

Documentation directive

The "`info/2`" documentation directive provides a way to declare arbitrary information about a predicate using both predefined and user-defined keywords.

info_directive →
    `:- info(` *predicate_indicator* `,` *info_list* `).`

info_list →
    `[]` |
    `[` *info_item* `is` *nonvar* `|` *info_list* `]`

info_item →
    `comment` | `author` | `version` | `date` | `parnames` |
    `argnames` | `definition` | `redefinition` | `allocation` |
    atom

When possible, one should use one of the above predefined information keywords instead of inventing our own. These predefined keywords may be especially processed by a Logtalk compiler, whenever automatic entity documentation is generated.

A.6 Clauses and goals

The syntax for goals and predicate clauses defined by the ISO Prolog Standard is extended to allow the use of Logtalk control constructs for message sending and for bypassing the Logtalk preprocessor.

A.6.1 Clauses

Object and category predicate clauses can be both regular Prolog clauses or clauses using the Definite clause grammar notation.

clauses →
    clause |
    clause clauses

clause →
    regular_clause |
    grammar_clause

grammar_clause →
    non_terminal `-->` *grammar_clause_body*

non_terminal →
    callable |
    non_terminal_message_to_self |
    non_terminal_message_to_object

non_terminal_message_to_object →
    receivers `::` *callable*

non_terminal_message_to_self →
    `::` *callable*

A.6.2 Goals

goal →
    callable |
    message_call |
    external_call

message_call →
    message_to_object |
    message_to_self |
    message_to_super

message_to_object →
    receivers `::` *messages*

message_to_self →
    `::` *messages*

message_to_super →
    `^^` *message*

messages →
    message |
    `(` *message* `,` *messages* `)` |
    `(` *message* `;` *messages* `)`

message →
    callable |
    variable

receivers →
    receiver |
    `(` *receiver* `,` *receivers* `)` |
    `(` *receiver* `;` *receivers* `)`

receiver →
    object_identifier |
    variable

external_call →
    `{` *callable* `}`

Note that, at compilation time, a message can be a non-instantiated variable. At runtime, the variable must be instantiated to a callable term before sending a message.

Otherwise, an exception will be generated. This is equivalent to sending the message `call(Variable)` to an object. Note that a message can be any callable term including control constructs like conjunctions, disjunctions, and if-then-else calls.

A.7 Entity properties

All entity types share the same set of properties:

category_property →
    `static` | `dynamic` |
    `built_in`

object_property →
    `static` | `dynamic` |
    `built_in`

protocol_property →
    `static` | `dynamic` |
    `built_in`

The property "`dynamic`" is often tested to ensure the applicability of built-in predicates and methods that modify or abolish dynamic entities. One example of a built-in entity is the pseudo-object "`user`" that contains all Prolog clauses that are not encapsulated by Logtalk objects or categories.

A.8 Predicate properties

The set of predicate properties defined in Logtalk is loosely based on ISO Standard for the Prolog module system. However, some properties have different meanings. For example, the properties "`public`" and "`private`" represent the predicate scope and not the possibility of inspecting the predicate source code.

predicate_property →
    `static` | `dynamic` |
    `private` | `protected` | `public` |
    `built_in` |
    `declared_in(` *entity_identifier* `)` |
    `defined_in(` *object_identifier* | *category_identifier* `)` |
    `metapredicate(` *metapredicate_mode_indicator* `)`

We can only retrieve the properties of declared predicates, built-in predicates, and built-in methods. In addition, note that some of the properties such as "`declared_in/1`" are only defined for user predicates.

Appendix B

Logtalk language reference

This appendix contains the Logtalk language reference. It includes a detailed description of every directive, built-in predicate, built-in method, and control construct defined in Logtalk. The descriptions are organized by groups (and sub-groups) and in alphabetical order inside each group. Note that most descriptions refer to terms defined in the Logtalk grammar, described in the previous appendix.

B.1 Directives

As seen in the appendix A, Logtalk defines two sets of directives: entity directives and predicate directives. Entity directives include directives for defining new entities and directives that affect the compilation of an entity as a whole. Predicate directives affect the compilation of specific predicates.

B.1.1 Entity directives

calls/1

Description

```
calls(Protocol)
calls(Protocol1, Protocol2, ...)
calls([Protocol1, Protocol2, ...])
```

This directive declares the protocols that are called by predicates defined in an object or category.

Templates and modes

```
calls(+protocol_identifiers)
```

Examples

```
:- calls(comparingp).
```

category/1-2

Description

```
category(Category)

category(Category,
    implements(Protocols))
```

Starting category directive.

Templates and modes

```
category(+category_identifier)

category(+category_identifier,
    implements(+implemented_protocols))
```

Examples

```
:- category(monitoring).

:- category(monitoring,
    implements(monitoringp)).

:- category(attributes,
    implements(protected::variables)).
```

dynamic/0

Description

```
dynamic
```

This directive declares an entity and all of its clauses dynamic.

Templates and modes

```
dynamic
```

Examples

```
:- dynamic.
```

end_category/0

Description

```
end_category
```

Ending category directive.

Templates and modes

```
end_category
```

Examples

```
:- end_category.
```

end_object/0

Description

```
end_object
```

Ending object directive.

Templates and modes

```
end_object
```

Examples

```
:- end_object.
```

end_protocol/0

Description

```
end_protocol
```

Ending protocol directive.

Templates and modes

```
end_protocol
```

Examples

```
:- end_protocol.
```

info/1

Description

```
info(List)
```

Documenting directive for objects, protocols, and categories.

Templates and modes

```
info(+info_list)
```

Examples

```
:- info([
    version is 1.0,
    author is 'Paulo Moura',
    date is 2000/4/20,
    comment is 'List protocol.']).
```

initialization/1

Description

```
initialization(Goal)
```

This directive sets up a goal to be called immediately after the container entity has been loaded to memory.

Templates and modes

```
initialization(@goal)
```

Examples

```
:- initialization(init).
```

object/1-5

Description

Stand-alone objects

```
object(Object)

object(Object,
    implements(Protocols))

object(Object,
    imports(Categories))

object(Object,
    implements(Protocols),
    imports(Categories))
```

Prototypes

```
object(Object,
    extends(Objects))

object(Object,
    implements(Protocols),
    extends(Objects))

object(Object,
    imports(Categories),
    extends(Objects))

object(Object,
    implements(Protocols),
    imports(Categories),
    extends(Objects))
```

Instances

```
object(Object,
    instantiates(Classes))

object(Object,
    implements(Protocols),
    instantiates(Classes))

object(Object,
    imports(Categories),
    instantiates(Classes))

object(Object,
    implements(Protocols),
    imports(Categories),
    instantiates(Classes))
```

Classes

```
object(Object,
    specializes(Classes))

object(Object,
    implements(Protocols),
    specializes(Classes))

object(Object,
    imports(Categories),
    specializes(Classes))

object(Object,
    implements(Protocols),
    imports(Categories),
    specializes(Classes))
```

Classes with metaclasses

```
object(Object,
    instantiates(Classes),
    specializes(Classes))

object(Object,
    implements(Protocols),
    instantiates(Classes),
    specializes(Classes))
```

```
object(Object,
    imports(Categories),
    instantiates(Classes),
    specializes(Classes))

object(Object,
    implements(Protocols),
    imports(Categories),
    instantiates(Classes),
    specializes(Classes))
```

Starting object directive.

Templates and modes

Stand-alone objects

```
object(+object_identifier)

object(+object_identifier,
    implements(+implemented_protocols))

object(+object_identifier,
    imports(+imported_categories))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories))
```

Prototypes

```
object(+object_identifier,
    extends(+extended_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    extends(+extended_objects))

object(+object_identifier,
    imports(+imported_categories),
    extends(+extended_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories),
    extends(+extended_objects))
```

Instances

```
object(+object_identifier,
    instantiates(+instantiated_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    instantiates(+instantiated_objects))

object(+object_identifier,
    imports(+imported_categories),
    instantiates(+instantiated_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories),
    instantiates(+instantiated_objects))
```

Classes

```
object(+object_identifier,
    specializes(+specialized_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    specializes(+specialized_objects))

object(+object_identifier,
    imports(+imported_categories),
    specializes(+specialized_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories),
    specializes(+specialized_objects))
```

Classes with metaclasses

```
object(+object_identifier,
    instantiates(+instantiated_objects),
    specializes(+specialized_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    instantiates(+instantiated_objects),
    specializes(+specialized_objects))
```

```
object(+object_identifier,
    imports(+imported_categories),
    instantiates(+instantiated_objects),
    specializes(+specialized_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories),
    instantiates(+instantiated_objects),
    specializes(+specialized_objects))
```

Examples

```
:- object(list).

:- object(list,
    implements(listp)).

:- object(list,
    extends(compound)).

:- object(list,
    implements(listp),
    extends(compound)).

:- object(object,
    imports(initialization),
    instantiates(class)).

:- object(abstract_class,
    instantiates(class),
    specializes(object)).

:- object(agent,
    imports(private::attributes)).
```

protocol/1-2

Description

```
protocol(Protocol)

protocol(Protocol,
    extends(Protocols))
```

Starting protocol directive.

Templates and modes

```
protocol(+protocol_identifier)
```

```
protocol(+protocol_identifier,
    extends(+extended_protocols))
```

Examples

```
:- protocol(listp).

:- protocol(listp,
    extends(compoundp)).

:- protocol(queuep,
    extends(protected::listp)).
```

uses/1

Description

```
uses(Object)
uses(Object1, Object2, ...)
uses([Object1, Object2, ...])
```

This directive declares objects that receive messages sent by predicates defined in the category or object containing the directive.

Templates and modes

```
uses(+object_identifiers)
```

Examples

```
:- uses(list).
```

B.1.2 Predicate directives

discontiguous/1

Description

```
discontiguous(Predicate)
discontiguous(Predicate1, Predicate2, ...)
discontiguous([Predicate1, Predicate2, ...])
```

This directive declares discontiguous predicates.

Templates and modes

```
discontiguous(+predicate_indicator_term)
```

Examples

```
:- discontiguous(counter/1).
:- discontiguous(lives/2, works/2).
:- discontiguous([db/4, key/2, file/3]).
```

dynamic/1

Description

```
dynamic(Predicate)
dynamic(Predicate1, Predicate2, ...)
dynamic([Predicate1, Predicate2, ...])
```

This directive declares dynamic predicates. Note that an object can be static and have both static and dynamic predicates.

Templates and modes

```
dynamic(+predicate_indicator_term)
```

Examples

```
:- dynamic(counter/1).
:- dynamic(lives/2, works/2).
:- dynamic([db/4, key/2, file/3]).
```

info/2

Description

```
info(Functor/Arity, List)
```

Documenting directive for predicates.

Templates and modes

```
info(+predicate_indicator, +info_list)
```

Examples

```
:- info(empty/1, [
    comment is 'True if the argument is an empty list.',
    argnames is ['List']]).
```

metapredicate/1

Description

```
metapredicate(Metapredicate)
```

This directive declares metapredicates, i.e. predicates that have arguments that will be called as goals.

Templates and modes

```
metapredicate(+metapredicate_predicate_term)
```

Examples

```
:- metapredicate(findall(*, ::, *)).
:- metapredicate(forall(::, ::)).
```

mode/2

Description

```
mode(Mode, Number_of_solutions)
```

Most predicates can be used with several instantiation modes. This directive enables the specification of each instantiation mode and the corresponding number of solutions/proofs.

Templates and modes

```
mode(+predicate_mode_term, +number_of_solutions)
```

Examples

```
:- mode(append(-, -, +), zero_or_more).
:- mode(append(+list, +list, -list), zero_or_one).
:- mode(var(@term), zero_or_one).
:- mode(arg(-, -, +), error).
```

op/3

Description

```
op(Precedence, Associativity, Operator)
```

Operator declaration.

Templates and modes

```
op(+integer, +associativity, +atom)
```

Examples

```
:- op(950, fx, +).
:- op(950, fx, ?).
:- op(950, fx, @).
:- op(950, fx, -).
```

private/1

Description

```
private(Predicate)
private(Predicate1, Predicate2, ...)
private([Predicate1, Predicate2, ...])
```

This directive declares private predicates. A private predicate can only be called from the object containing the private directive.

Templates and modes

```
private(+predicate_indicator_term)
```

Examples

```
:- private(counter/1).
:- private(init/1, free/1).
:- private([data/3, key/1, keys/1]).
```

protected/1

Description

```
protected(Predicate)
protected(Predicate1, Predicate2, ...)
protected([Predicate1, Predicate2, ...])
```

This directive declares protected predicates. A protected predicate can only be called from the object containing the declaration, or from an object that inherits the declaration.

Templates and modes

```
protected(+predicate_indicator_term)
```

Examples

```
:- protected(init/1).
:- protected(print/2, convert/4).
:- protected([load/1, save/3]).
```

public/1

Description

```
public(Predicate)
public(Predicate1, Predicate2, ...)
public([Predicate1, Predicate2, ...])
```

This directive declares public predicates. A public predicate can be called from any object.

Templates and modes

```
public(+predicate_indicator_term)
```

Examples

```
:- public(ancestor/1).
:- public(instance/1, instances/1).
:- public([leaf/1, leaves/1]).
```

B.2 Built-in predicates

Logtalk adds a new set of built-in predicates to the ISO Prolog standard. These predicates, described in this section, enable the programmer to inspect, create, and abolish Logtalk entities, compile and load Logtalk source files, and inspect, create, and abolish events.

B.2.1 Enumerating entities

This set of built-in predicates enumerate, by backtracking, defined objects, protocols, and categories.

current_category/1

Description

`current_category(Category)`

This built-in predicate enumerates, by backtracking, all currently defined categories. All categories are returned, static, dynamic, or built-in.

Templates and modes

`current_category(?category_identifier)`

Errors

Category is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`

Examples

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

current_object/1

Description

`current_object(Object)`

This built-in predicate enumerates, by backtracking, all currently defined objects. All objects are returned, static, dynamic or built-in.

Templates and modes

`current_object(?object_identifier)`

Errors

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`

Examples

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

current_protocol/1

Description

`current_protocol(Protocol)`

This built-in predicate enumerates, by backtracking, all currently defined protocols. All protocols are returned, static, dynamic, or built-in.

Templates and modes

`current_protocol(?protocol_identifier)`

Errors

Protocol is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`

Examples

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

B.2.2 Enumerating entity properties

This section describes built-in predicates for enumerating, by backtracking, object, protocol, and category properties.

category_property/2

Description

```
category_property(Category, Property)
```

This built-in predicate enumerates, by backtracking, the properties of defined categories.

Templates and modes

```
category_property(?category_identifier, ?category_property)
```

Errors

Category is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`
Property is neither a variable nor a valid category property:
`domain_error(category_property, Property)`

Examples

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

object_property/2

Description

```
object_property(Object, Property)
```

This built-in predicate enumerates, by backtracking, the properties of defined objects.

Templates and modes

```
object_property(?object_identifier, ?object_property)
```

Errors

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Property is neither a variable nor a valid object property:
`domain_error(object_property, Property)`

Examples

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

protocol_property/2

Description

```
protocol_property(Protocol, Property)
```

This built-in predicate enumerates, by backtracking, the properties of defined protocols.

Templates and modes

```
protocol_property(?protocol_identifier, ?protocol_property)
```

Errors

Protocol is neither a variable nor a valid protocol identifier:

```
type_error(protocol_identifier, Protocol)
```

Property is neither a variable nor a valid protocol property:

```
domain_error(protocol_property, Property)
```

Examples

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

B.2.3 Creating new entities

This section describes built-in predicates that enable the dynamic creation of new objects, protocols, and categories.

create_category/4

Description

```
create_category(Identifier, Relations, Directives, Clauses)
```

This built-in predicate creates a new dynamic category.

Templates and modes

```
create_category(+category_identifier, +list, +list, +list)
```

Errors

Identifier is a variable:

```
instantiation_error
```

Identifier is not a valid category identifier:

```
type_error(category_identifier, Identifier)
```

Identifier is already in use:

```
permission_error(replace, category, Identifier)
permission_error(replace, object, Identifier)
permission_error(replace, protocol, Identifier)
```

Relations is not a list:

```
type_error(list, Relations)
```

Directives is not a list:

```
type_error(list, Directives)
```

Clauses is not a list:

```
type_error(list, Clauses)
```

Examples

```
| ?- create_category(foo, [implements(barp)], [], [bar(foo)]).
```

create_object/4

Description

```
create_object(Identifier, Relations, Directives, Clauses)
```

This built-in predicate creates a new dynamic object.

Templates and modes

```
create_object(+object_identifier, +list, +list, +list)
```

Errors

Identifier is a variable:
```
instantiation_error
```
Identifier is not a valid object identifier:
```
type_error(object_identifier, Identifier)
```
Identifier is already in use:
```
permission_error(replace, category, Identifier)
permission_error(replace, object, Identifier)
permission_error(replace, protocol, Identifier)
```
Relations is not a list:
```
type_error(list, Relations)
```
Directives is not a list:
```
type_error(list, Directives)
```
Clauses is not a list:
```
type_error(list, Clauses)
```

Examples

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

create_protocol/3

Description

```
create_protocol(Identifier, Relations, Directives)
```

This built-in predicate creates a new dynamic protocol.

Templates and modes

```
create_protocol(+protocol_identifier, +list, +list)
```

Errors

Identifier is a variable:
```
instantiation_error
```
Identifier is not a valid protocol identifier:
```
type_error(protocol_identifier, Identifier)
```

Identifier is already in use:
```
permission_error(replace, category, Identifier)
permission_error(replace, object, Identifier)
permission_error(replace, protocol, Identifier)
```
Relations is not a list:
```
type_error(list, Relations)
```
Directives is not a list:
```
type_error(list, Directives)
```

Examples

```
| ?- create_protocol(foop, [extends(barp)], [public(foo/1)]).
```

B.2.4 Abolishing entities

This section describes the built-in predicates that enable the abolishing of defined dynamic entities (objects, protocols, or categories).

abolish_category/1

Description

```
abolish_category(Category)
```

This built-in predicate removes a dynamic category from the database.

Templates and modes

```
abolish_category(@category_identifier)
```

Errors

Category is a variable:
```
instantiation_error
```
Category is not a valid category identifier:
```
type_error(category_identifier, Category)
```
Category is an identifier of a static category:
```
permission_error(modify, static_category, Category)
```
Category does not exist:
```
existence_error(category, Category)
```

Examples

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

abolish_object/1

Description

```
abolish_object(Object)
```

This built-in predicate removes a dynamic object from the database.

Templates and modes

```
abolish_object(@object_identifier)
```

Errors

Object is a variable:
```
instantiation_error
```
Object is not a valid object identifier:
```
type_error(object_identifier, Object)
```
Object is an identifier of a static object:
```
permission_error(modify, static_object, Object)
```
Object does not exist:
```
existence_error(object, Object)
```

Examples

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

abolish_protocol/1

Description

```
abolish_protocol(Protocol)
```

This built-in predicate removes a dynamic protocol from the database.

Templates and modes

```
abolish_protocol(@protocol_identifier)
```

Errors

Protocol is a variable:
```
instantiation_error
```
Protocol is not a valid protocol identifier:
```
type_error(protocol_identifier, Protocol)
```
Protocol is an identifier of a static protocol:
```
permission_error(modify, static_protocol, Protocol)
```
Protocol does not exist:
```
existence_error(protocol, Protocol)
```

Examples

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

B.2.5 Entity relations

This section describes a set of built-in predicates that enumerate, by backtracking, the relations between objects, protocols, and categories.

extends_object/2-3

Description

```
extends_object(Prototype, Parent)
extends_object(Prototype, Parent, Scope)
```

This built-in predicate enumerates, by backtracking, all pairs of objects such that the first one extends the second.

Templates and modes

```
extends_object(?object_identifier, ?object_identifier)
extends_object(?object_identifier, ?object_identifier, ?scope)
```

Errors

Prototype is neither a variable nor a valid object identifier:
`type_error(object_identifier, Prototype)`
Parent is neither a variable nor a valid object identifier:
`type_error(object_identifier, Parent)`
Scope is neither a variable nor a valid entity relation scope:
`type_error(scope, Scope)`

Examples

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

| ?- extends_object(Object, list, public).
```

extends_protocol/2-3

Description

```
extends_protocol(Protocol1, Protocol2)
extends_protocol(Protocol1, Protocol2, Scope)
```

This built-in predicate enumerates, by backtracking, all pairs of protocols such that the first one extends the second.

Templates and modes

```
extends_protocol(?protocol_identifier, ?protocol_identifier)
extends_protocol(?protocol_identifier, ?protocol_identifier, ?scope)
```

Errors

Protocol1 is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol1)`
Protocol2 is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol2)`
Scope is neither a variable nor a valid entity relation scope:
`type_error(scope, Scope)`

Examples

```
| ?- extends_protocol(listp, Protocol).

| ?- extends_protocol(Protocol, termp, private).
```

implements_protocol/2-3

Description

```
implements_protocol(Object, Protocol)
implements_protocol(Category, Protocol)

implements_protocol(Object, Protocol, Scope)
implements_protocol(Category, Protocol, Scope)
```

This built-in predicate enumerates, by backtracking, all pairs of entities such that an object or a category implements a protocol.

Templates and modes

```
implements_protocol(?object_identifier, ?protocol_identifier)
implements_protocol(?category_identifier, ?protocol_identifier)

implements_protocol(?object_identifier, ?protocol_identifier, ?scope)
implements_protocol(?category_identifier, ?protocol_identifier, ?scope)
```

Errors

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Category is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`
Protocol is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`
Scope is neither a variable nor a valid entity relation scope:
`type_error(scope, Scope)`

Examples

```
| ?- implements_protocol(List, listp).

| ?- implements_protocol(List, listp, public).
```

imports_category/2-3

Description

```
imports_category(Object, Category)
imports_category(Object, Category, Scope)
```

This built-in predicate enumerates, by backtracking, all pairs of objects and categories such that the first one imports the other.

Templates and modes

```
imports_category(?object_identifier, ?category_identifier)
imports_category(?object_identifier, ?category_identifier, ?scope)
```

Errors

Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Category is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`
Scope is neither a variable nor a valid entity relation scope:
`type_error(scope, Scope)`

Examples

```
| ?- imports_category(debugger, monitoring).

| ?- imports_category(Object, monitoring, protected).
```

instantiates_class/2-3

Description

```
instantiates_class(Instance, Class)
instantiates_class(Instance, Class, Scope)
```

This built-in predicate enumerates, by backtracking, all pairs of objects such that the first one instantiates the second.

Templates and modes

```
instantiates_class(?object_identifier, ?object_identifier)
instantiates_class(?object_identifier, ?object_identifier, ?scope)
```

Errors

Instance is neither a variable nor a valid object identifier:
`type_error(object_identifier, Instance)`
Class is neither a variable nor a valid object identifier:
`type_error(object_identifier, Class)`
Scope is neither a variable nor a valid entity relation scope:
`type_error(scope, Scope)`

Examples

```
| ?- instantiates_class(water_jug, state_space).

| ?- instantiates_class(Space, state_space, public).
```

specializes_class/2-3

Description

```
specializes_class(Class, Superclass)
specializes_class(Class, Superclass, Scope)
```

This built-in predicate enumerates, by backtracking, all pairs of objects such that the first one specializes the second.

Templates and modes

```
specializes_class(?object_identifier, ?object_identifier)
specializes_class(?object_identifier, ?object_identifier, ?scope)
```

Errors

Class is neither a variable nor a valid object identifier:
`type_error(object_identifier, Class)`
Superclass is neither a variable nor a valid object identifier:
`type_error(object_identifier, Superclass)`
Scope is neither a variable nor a valid entity relation scope:
`type_error(scope, Scope)`

Examples

```
| ?- specializes_class(Subclass, state_space).

| ?- specializes_class(Subclass, state_space, public).
```

B.2.6 Event handling

This section describes the set of built-in predicates that enable the definition, inspection, and abolishing of events.

abolish_events/5

Description

```
abolish_events(Event, Object, Message, Sender, Monitor)
```

Abolishes all matching events.

Templates and modes

```
abolish_events(@event, @object_identifier, @callable,
               @object_identifier, @object_identifier)
```

Errors

Event is neither a variable nor a valid event identifier:
`type_error(event, Event)`
Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Message is neither a variable nor a callable term:
`type_error(callable, Message)`
Sender is neither a variable nor a valid object identifier:
`type_error(object_identifier, Sender)`
Monitor is neither a variable nor a valid object identifier:
`type_error(object_identifier, Monitor)`

Examples

```
| ?- abolish_events(_, list, _, _, debugger).
```

current_event/5

Description

```
current_event(Event, Object, Message, Sender, Monitor)
```

This built-in predicate enumerates, by backtracking, all defined events.

Templates and modes

```
current_event(?event, ?object_identifier, ?callable,
              ?object_identifier, ?object_identifier)
```

Errors

Event is neither a variable nor a valid event identifier:
`type_error(event, Event)`
Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Message is neither a variable nor a callable term:
`type_error(callable, Message)`
Sender is neither a variable nor a valid object identifier:
`type_error(object_identifier, Sender)`
Monitor is neither a variable nor a valid object identifier:
`type_error(object_identifier, Monitor)`

Examples

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

define_events/5

Description

```
define_events(Event, Object, Message, Sender, Monitor)
```

This built-in predicate defines a new set of events.

Templates and modes

```
define_events(@event, @object_identifier, @callable,
              @object_identifier, +object_identifier)
```

Errors

Event is neither a variable nor a valid event identifier:
`type_error(event, Event)`
Object is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
Message is neither a variable nor a callable term:
`type_error(callable, Message)`
Sender is neither a variable nor a valid object identifier:
`type_error(object_identifier, Sender)`
Monitor is a variable:
`instantiation_error`
Monitor is neither a variable nor a valid object identifier:
`type_error(object_identifier, Monitor)`

Examples

```
| ?- define_events(_, list, member(_, _), _ , debugger).
```

B.2.7 Compiling and loading entities

Compiling and loading source files is usually regarded as an implementation-dependent feature and not part of a formal language definition. However, when we are compiling and loading a source file, the only thing that is indeed implementation-dependent, is the for filenames and pathnames syntax: the language specification already describes what is correct code!

logtalk_compile/1

Description

```
logtalk_compile(Entity)
logtalk_compile(Entities)
```

This built-in predicate compiles to disk an entity or a list of entities (objects, protocols, or categories) using the default compiler options specified in the Logtalk configuration file. The Logtalk file name extension (by default, "`.lgt`") must be omitted. Note that the argument is a list of entity/file names, not file paths.

Templates and modes

```
logtalk_compile(@atom_or_atom_list)
```

Errors

Entity is a variable:
```
instantiation_error
```
Entities is a variable or a list with an element which is a variable:
```
instantiation_error
```
Entities is neither a variable nor an atom nor a proper list:
```
type_error(list, Entities)
```
An element Entity of the Entities list is neither a variable nor an atom:
```
type_error(atom, Entity)
```
Entity or an element Entity of the Entities list does not exist in the current working directory:
```
existence_error(entity, Entity)
```

Examples

```
| ?- logtalk_compile(tree).
| ?- logtalk_compile([listp, list]).
```

logtalk_compile/2

Description

```
logtalk_compile(Entity, Options)
logtalk_compile(Entities, Options)
```

This built-in predicate compiles to disk an entity or a list of entities (objects, protocols, or categories) using a list of options. The Logtalk file name extension (by default, "`.lgt`") must be omitted. Note that the first argument is a list of entity/file names, not file paths.

Templates and modes

```
logtalk_compile(@atom_or_atom_list, @list)
```

Errors

Entity is a variable:
`instantiation_error`
Entities is a variable or a list with an element which is a variable:
`instantiation_error`
Entities is neither a variable nor an atom nor a proper list:
`type_error(list, Entities)`
An element Entity of the Entities list is neither a variable nor an atom:
`type_error(atom, Entity)`
Entity or an element Entity of the Entities list does not exist in the current working directory:
`existence_error(entity, Entity)`
Options is a variable:
`instantiation_error`
Options is neither a variable nor a proper list:
`type_error(list, Options)`
An element Option of the Options list is not valid:
`type_error(compiler_option, Option)`

Examples

```
| ?- logtalk_compile(list, []).

| ?- logtalk_compile([listp, list], [xml(off), report(on)]).
```

logtalk_load/1

Description

```
logtalk_load(Entity)
logtalk_load(Entities)
```

This built-in predicate compiles to disk and then loads to memory an entity or a list of entities (objects, protocols or categories) using the default compiler options specified in the Logtalk configuration file. The Logtalk file name extension (by default, "`.lgt`") must be omitted. Note that the argument is a list of entity/file names, not file paths.

Templates and modes

```
logtalk_load(@atom_or_atom_list)
```

Errors

Entity is a variable:
`instantiation_error`
Entities is a variable or a list with an element which is a variable:
`instantiation_error`
Entities is neither a variable nor an atom nor a proper list:
`type_error(list, Entities)`
An element Entity of the Entities list is neither a variable nor an atom:
`type_error(atom, Entity)`
Entity or an element Entity of the Entities list does not exist in the current working directory:
`existence_error(entity, Entity)`

Examples

```
| ?- logtalk_load(tree).

| ?- logtalk_load([listp, list]).
```

logtalk_load/2

Description

```
logtalk_load(Entity, Options)
logtalk_load(Entities, Options)
```

This built-in predicate compiles to disk and then loads to memory an entity or a list of entities (objects, protocols or categories) using a list of options. The Logtalk file name extension (by default, "`.lgt`") must be omitted. Note that the first argument is a list of entity/file names, not file paths.

Templates and modes

```
logtalk_load(@atom_or_atom_list, @list)
```

Errors

Entity is a variable:
`instantiation_error`
Entities is a variable or a list with an element which is a variable:
`instantiation_error`
Entities is neither a variable nor an atom nor a proper list:
`type_error(list, Entities)`
An element Entity of the Entities list is neither a variable nor an atom:
`type_error(atom, Entity)`
Entity or an element Entity of the Entities list does not exist in the current working directory:
`existence_error(entity, Entity)`
Options is a variable:
`instantiation_error`
Options is neither a variable nor a proper list:
`type_error(list, Options)`

An element Option of the Options list is not valid:
`type_error(compiler_option, Option)`

Examples

```
| ?- logtalk_load(list, []).

| ?- logtalk_load([listp, list], [xml(off), report(on)]).
```

B.2.8 Flags

At runtime, the current compiler flags or options can be consulted and changed by calling the built-in predicates `current_logtalk_flag/2` and `set_logtalk_flag/2`. The initial set of compiler options is read from the Logtalk configuration file.

current_logtalk_flag/2

Description

`current_logtalk_flag(Flag, Value)`

Enumerates, by backtracking, the current Logtalk flag values.

Templates and modes

`current_logtalk_flag(?atom, ?atom)`

Errors

Flag is neither a variable nor an atom:
`type_error(atom, Flag)`
Flag is not a valid flag:
`domain_error(valid_flag, Value)`

Examples

```
| ?- current_logtalk_flag(xml, Value).
```

set_logtalk_flag/2

Description

`set_logtalk_flag(Flag, Value)`

Sets Logtalk flag values.

Templates and modes

`set_logtalk_flag(+atom, +atom)`

Errors

Flag is a variable:
`instantiation_error`
Value is a variable:
`instantiation_error`

Flag is not an atom:
`type_error(atom, Flag)`
Flag is neither a variable nor a valid flag:
`domain_error(valid_flag, Flag)`
Value is not a valid value for flag Flag:
`domain_error(valid_flag_value, Value)`
Flag is a read-only flag:
`domain_error(read_only_flag, Flag)`

Examples

```
| ?- set_logtalk_flag(xml, on).
```

B.2.9 Others

This is a catch-all group for some useful or common predicates.

forall/2

Description

`forall(Generator, Test)`

This predicate is true if Test is true for all Generator solutions (some Prolog compilers already define this predicate or a similar predicate).

Templates and modes

`forall(+callable, +callable)`

Errors

Generator is not a callable term:
`type_error(callable, Generator)`
Test is not a callable term:
`type_error(callable, Test)`

Examples

```
| ?- forall(member(X, [1, 2, 3]), write(X)).
```

retractall/1

Description

`retractall(Head)`

Logtalk adds this built-in predicate, with the usual definition, to a Prolog compiler that does not defines it.

Templates and modes

`retractall(+callable)`

Errors

Head is not a callable term:
`type_error(callable, Head)`

Examples

```
| ?- retractall(foo(_)).
```

B.3 Built-in methods

Logtalk defines a set of built-in object predicates or methods that are preprocessed by the Logtalk compiler.

B.3.1 Local methods

Local methods are methods that can only be called inside an object, providing access to the message execution context.

parameter/2

Description

```
parameter(Number, Term)
```

This method is used only in parametric objects. It returns parameter values by using the parameter position in the entity identifier. Note that this predicate implies an instantiation between its second argument and the corresponding implicit context argument in the predicate containing the call. This instantiation occurs at the clause head, not at the clause body. See also `this/1`.

Templates and modes

```
parameter(+integer, ?term)
```

Errors

Number is a variable:
`instantiation_error`
Number is neither a variable nor an integer value:
`type_error(integer, Number)`
Object identifier is not a compound term:
`type_error(compound, Object)`
Number is a negative integer value:
`domain_error(not_less_than_zero, Number)`

Examples

```
:- object(box(_Colour)).
    ...
    colour(Colour) :-
        parameter(1, Colour).
        ...
    ...
```

self/1

Description

```
self(Self)
```

This built-in method returns the object (*self*) which has received the message under processing. Note that this predicate implies an instantiation between its argument and the corresponding implicit context argument in the predicate containing the call. This instantiation occurs at the clause head, not at the clause body.

Templates and modes

```
self(?object_identifier)
```

Errors

(none)

Examples

```
test :-
    self(Self),
    write('executing a method in behalf of '),
    writeq(Self), nl.
```

sender/1

Description

```
sender(Sender)
```

This built-in method returns the object (*sender*) which has sent the message under processing. Note that this predicate implies an instantiation between its argument and the corresponding implicit context argument in the predicate containing the call. This instantiation occurs at the clause head, not at the clause body.

Templates and modes

```
sender(-object_identifier)
```

Errors

(none)

Examples

```
test :-
    sender(Sender),
    write('executing a method to answer a message sent by '),
    writeq(Sender), nl.
```

this/1

Description

```
this(This)
```

This built-in method unifies its argument with the identifier of the object containing the predicate definition under execution ((*this*)). Note that this predicate implies an instantiation between its argument and the corresponding implicit context argument in the predicate containing the call. This instantiation occurs at the clause head, not at the clause body. This method is useful in avoiding problems whenever an object is renamed or whenever parametric objects are used. It can also be used to retrieve runtime parameters of parametric object though unification (see also `parameter/2`).

Templates and modes

```
this(?object_identifier)
```

Errors

(none)

Examples

```
test :-
    this(This),
    write('executing a definition contained in '),
    writeq(This), nl.
```

B.3.2 Reflection methods

This section describes two predicates which can be used to enumerate object predicates and object predicate properties.

current_predicate/1

Description

```
current_predicate(Predicate)
```

This built-in method enumerates, by backtracking, the visible user predicates for an object.

Templates and modes

```
current_predicate(?predicate_indicator)
```

Errors

Predicate is neither a variable nor a valid predicate indicator:

```
type_error(predicate_indicator, Predicate)
```

Examples

To enumerate, by backtracking, the user predicates visible in *this*:

```
current_predicate(Predicate)
```

To enumerate, by backtracking, the public and protected user predicates visible in *self*:

```
::current_predicate(Predicate)
```

To enumerate, by backtracking, the public user predicates visible for an object:

```
Object::current_predicate(Predicate)
```

predicate_property/2

Description

```
predicate_property(Predicate, Property)
```

This built-in method enumerates, by backtracking, the properties of a visible predicate.

Templates and modes

```
predicate_property(+callable, ?predicate_property)
```

Errors

Predicate is a variable:
`instantiation_error`
Predicate is neither a variable nor a callable term:
`type_error(callable, Predicate)`
Property is neither a variable nor a valid predicate property:
`domain_error(predicate_property, Property)`

Examples

To enumerate, by backtracking, the properties of a predicate visible in *this*:
`predicate_property(foo(_), Property)`
To enumerate, by backtracking, the properties of a public or protected predicate visible in *self*:
`::predicate_property(foo(_), Property)`
To enumerate, by backtracking, the properties of a public predicate visible in an object:
`Object::predicate_property(foo(_), Property)`

B.3.3 Database methods

The database built-in methods perform functions similar to the built-in Prolog predicates with the same name, acting on an object instead of the whole database.

abolish/1

Description

```
abolish(Predicate)
abolish(Functor/Arity)
```

This built-in method removes a runtime declared dynamic predicate from an object database.

Templates and modes

```
abolish(+predicate_indicator)
```

Errors

Predicate is a variable:
```
instantiation_error
```
Predicate is neither a variable nor a valid predicate indicator:
```
type_error(predicate_indicator, Predicate)
```
Functor is neither a variable nor an atom:
```
type_error(atom, Functor)
```
Arity is neither a variable nor an integer:
```
type_error(integer, Arity)
```
Predicate is statically declared:
```
permission_error(modify, predicate_declaration, Functor/Arity)
```
Predicate is a private predicate:
```
permission_error(modify, private_predicate, Functor/Arity)
```
Predicate is a protected predicate:
```
permission_error(modify, protected_predicate, Functor/Arity)
```
Predicate is a static predicate:
```
permission_error(modify, static_predicate, Functor/Arity)
```
Predicate is not declared or is declared but not in the object receiving the message::
```
existence_error(predicate_declaration, Functor/Arity)
```

Examples

To abolish any dynamic predicate in *this*:
```
abolish(Predicate)
```
To abolish a public or protected dynamic predicate in *self*:
```
::abolish(Predicate)
```
To abolish a public dynamic predicate in an object:
```
Object::abolish(Predicate)
```

asserta/1

Description

```
asserta(Clause)
asserta((Head:-Body))
```

This built-in method asserts a clause as the first one for an object's dynamic predicate. If the predicate has not yet been declared, then a dynamic predicate declaration is added to the object.

Templates and modes

```
asserta(+clause)
```

Errors

Clause is a variable:
```
instantiation_error
```
Head is a variable:
```
instantiation_error
```
Head is neither a variable nor a callable term:
```
type_error(callable, Head)
```

Body cannot be converted to a goal:
`type_error(callable, Body)`
The predicate indicator of Head is that of a private predicate:
`permission_error(modify, private_predicate, Head)`
The predicate indicator of Head is that of a protected predicate:
`permission_error(modify, protected_predicate, Head)`
The predicate indicator of Head is that of a static predicate:
`permission_error(modify, static_predicate, Head)`

Examples

To assert a clause as the first one for any dynamic predicate in *this*:
`asserta(Clause)`
To assert a clause as the first one for any public or protected dynamic predicate in *self*:
`::asserta(Clause)`
To assert a clause as the first one for any public dynamic predicate in an object:
`Object::asserta(Clause)`

assertz/1

Description

```
assertz(Clause)
assertz((Head:-Body))
```

This built-in method asserts a clause as the last one for an object's dynamic predicate. If the predicate has not yet been declared, then a dynamic predicate declaration is added to the object.

Templates and modes

```
assertz(+clause)
```

Errors

Clause is a variable:
`instantiation_error`
Head is a variable:
`instantiation_error`
Head is neither a variable nor a callable term:
`type_error(callable, Head)`
Body cannot be converted to a goal:
`type_error(callable, Body)`
The predicate indicator of Head is that of a private predicate:
`permission_error(modify, private_predicate, Head)`
The predicate indicator of Head is that of a protected predicate:
`permission_error(modify, protected_predicate, Head)`
The predicate indicator of Head is that of a static predicate:
`permission_error(modify, static_predicate, Head)`

Examples

To assert a clause as the last one for any dynamic predicate in *this*:
`assertz(Clause)`

To assert a clause as the last one for any public or protected dynamic predicate in *self*:

```
::assertz(Clause)
```

To assert a clause as the last one for any public dynamic predicate in an object:

```
Object::assertz(Clause)
```

clause/2

Description

```
clause(Head, Body)
```

This built-in method enumerates, by backtracking, the clauses of an object's dynamic predicates.

Templates and modes

```
clause(+callable, ?body)
```

Errors

Head is a variable:

```
instantiation_error
```

Head is neither a variable nor a callable term:

```
type_error(callable, Head)
```

Body is neither a variable nor a callable term:

```
type_error(callable, Body)
```

The predicate indicator of Head is that of a private predicate:

```
permission_error(access, private_predicate, Head)
```

The predicate indicator of Head is that of a protected predicate:

```
permission_error(access, protected_predicate, Head)
```

The predicate indicator of Head is that of a static predicate:

```
permission_error(access, static_predicate, Head)
```

Head is not a declared predicate:

```
existence_error(predicate_declaration, Head)
```

Examples

To retrieve a matching clause of any dynamic predicate in *this*:

```
clause(Head, Body)
```

To retrieve a matching clause of a public or protected dynamic predicate in *self*:

```
::clause(Head, Body)
```

To retrieve a matching clause of a public dynamic predicate in an object:

```
Object::clause(Head, Body)
```

retract/1

Description

```
retract(Clause)
retract((Head:-Body))
```

This built-in method retracts a dynamic clause from an object.

Templates and modes

```
retract(+clause)
```

Errors

Head is a variable:
`instantiation_error`
Head is neither a variable nor a callable term:
`type_error(callable, Head)`
The predicate indicator of Head is that of a private predicate:
`permission_error(modify, private_predicate, Head)`
The predicate indicator of Head is that of a protected predicate:
`permission_error(modify, protected_predicate, Head)`
The predicate indicator of Head is that of a static predicate:
`permission_error(modify, static_predicate, Head)`
The predicate indicator of Head is not declared:
`existence_error(predicate_declaration, Head)`

Examples

To retract a matching clause of any dynamic predicate in *this*:
`retract(Clause)`
To retract a matching clause of a public or protected dynamic predicate in *self*:
`::retract(Clause)`
To retract a matching clause of a public dynamic predicate in an object:
`Object::retract(Clause)`

retractall/1

Description

```
retractall(Head)
```

This built-in method retracts all matching predicates from an object.

Templates and modes

```
retractall(+callable)
```

Errors

Head is a variable:
`instantiation_error`
Head is neither a variable nor a callable term:
`type_error(callable, Head)`
The predicate indicator of Head is that of a private predicate:
`permission_error(modify, private_predicate, Head)`
The predicate indicator of Head is that of a protected predicate:
`permission_error(modify, protected_predicate, Head)`
The predicate indicator of Head is that of a static predicate:
`permission_error(modify, static_predicate, Head)`
The predicate indicator of Head is not declared:
`existence_error(predicate_declaration, Head)`

Examples

To retract all matching predicate definitions in *this*:

```
retractall(Head)
```

To retract all matching public or protected predicate definitions in *self*:

```
::retractall(Head)
```

To retract all matching public predicate definitions in an object:

```
Object::retractall(Head)
```

B.3.4 All solutions methods

This set of built-in methods act on an object database, performing similar functions to the built-in Prolog predicates with the same name.

bagof/3

Description

```
bagof(Term, Goal, List)
```

(see the Prolog ISO standard definition)

Templates and modes

```
bagof(@term, +callable, -list)
```

Errors

(see the Prolog ISO standard)

Examples

To find all solutions in *this*:

```
bagof(Term, Goal, List)
```

To find all solutions in *self*:

```
bagof(Term, ::Goal, List)
```

To find all solutions in an object:

```
bagof(Term, Object::Goal, List)
```

findall/3

Description

```
findall(Term, Goal, List)
```

(see the Prolog ISO standard definition)

Templates and modes

```
findall(@term, +callable, -list)
```

Errors

(see the Prolog ISO standard)

Examples

To find all solutions in *this*:

```
findall(Term, Goal, List)
```

To find all solutions in *self*:

```
findall(Term, ::Goal, List)
```

To find all solutions in an object:

```
findall(Term, Object::Goal, List)
```

forall/2

Description

```
forall(Generator, Test)
```

For all solutions of Generator, Test is true.

Templates and modes

```
forall(+callable, +callable)
```

Errors

Generator is a variable:

```
instantiation_error
```

Test is a variable:

```
instantiation_error
```

Generator is neither a variable nor a callable term:

```
type_error(callable, Generator)
```

Test is neither a variable nor a callable term:

```
type_error(callable, Test)
```

Examples

To call both goals in *this*:

```
forall(Generator, Test)
```

To call both goals in *self*:

```
forall(::Generator, ::Test)
```

To call both goals in an object:

```
forall(Object::Generator, Object::Test)
```

setof/3

Description

```
setof(Term, Goal, List)
```

(see the Prolog ISO standard definition)

Templates and modes

```
setof(@term, +callable, -list)
```

Errors

(see the Prolog ISO standard)

Examples

To find all solutions in *this*:

```
setof(Term, Goal, List)
```

To find all solutions in *self*:

```
setof(Term, ::Goal, List)
```

To find all solutions in an object:

```
setof(Term, Object::Goal, List)
```

B.3.5 Event handler methods

Logtalk declares (but does not define) two event handler methods. Each object, acting as a monitor, should define (or inherit definitions for) these predicates.

before/3

Description

```
before(Object, Message, Sender)
```

This is a predeclared, though user-defined, public method for handling *before* events.

Templates and modes

```
before(?object, ?term, ?object)
```

Errors

(none)

Examples

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

after/3

Description

```
after(Object, Message, Sender)
```

This is a predeclared, though user-defined, public method for handling *after* events.

Templates and modes

```
after(?object, ?term, ?object)
```

Errors

(none)

Examples

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

B.3.6 Definite clause grammar parsing methods

phrase/2

Description

`phrase(Ruleset, Input)`

Templates and modes

`phrase(+callable, ?list)`

True if the list `Input` can be parsed using the specified `Ruleset`.

Errors

Ruleset is a variable:
`instantiation_error`
Ruleset is neither a variable nor a callable term:
`type_error(callable, Ruleset)`
Input is neither a variable nor a proper list:
`type_error(list, Input)`

Examples

(none)

phrase/3

Description

`phrase(Ruleset, Input, Rest)`

True if the list `Input` can be parsed using the specified `Ruleset`. The list `Rest` is what remains of the list `Input` after parsing succeeded.

Templates and modes

`phrase(+callable, ?list, ?list)`

Errors

Ruleset is a variable:
`instantiation_error`
Ruleset is neither a variable nor a callable term:
`type_error(callable, Ruleset)`
Input is neither a variable nor a proper list:
`type_error(list, Input)` Rest is neither a variable nor a proper list:
`type_error(list, Rest)`

Examples

(none)

B.4 Control constructs

Logtalk adds four new control constructs to the ISO Prolog standard. These new constructs, described below, are used to send messages and to bypass the Logtalk preprocessor.

B.4.1 Message sending

`::/2`

Description

```
Object::Predicate

(Object1, Object2, ...)::Predicate
(Object1; Object2; ...)::Predicate

Object::(Predicate1, Predicate2, ...)
Object::(Predicate1; Predicate2; ...)
```

This control construct implements the sending of a message to an object. The message argument must match a public predicate of the receiver object. We can also send the same message to a set of objects or send a set of messages to a single object. The "," and ";" in the list have the usual Prolog meaning.

Templates and modes

```
+receivers::+messages
```

Errors

Either Object or Predicate is a variable:
`instantiation_error`
Predicate is declared private:
`permission_error(access, private_predicate, Predicate)`
Predicate is declared protected:
`permission_error(access, protected_predicate, Predicate)`
Predicate is not declared:
`existence_error(predicate_declaration, Predicate)`
Object does not exist:
`existence_error(object, Object)`

Examples

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

`::/1`

Description

```
::Predicate
::(Predicate1, Predicate2, ...)
::(Predicate1; Predicate2; ...)
```

This control construct implements the sending of a message to *self*. It is only used in the body of a predicate definition. The argument should match a public or protected predicate of *self*. It may also match a private predicate whenever the predicate is imported from a category, used in a category, or is inherited using private inheritance. We can also send a set of messages to *self*. The "," and ";" in the list have the usual Prolog meaning.

Templates and modes

```
::+messages
```

Errors

Predicate is a variable:
```
instantiation_error
```
Predicate is declared private:
```
permission_error(access, private_predicate, Predicate)
```
Predicate is not declared:
```
existence_error(predicate_declaration, Predicate)
```

Examples

```
area(Area) :-
    ::width(Width),
    ::height(Height),
    Area is Width*Height.
```

`^^/1`

Description

```
^^Predicate
```

This control construct implements calling of a redefined/inherited definition for a message. Usually, it is only used in the body of a predicate definition. The predicate must match a public or protected predicate of *self* or be within the scope of *this*.

Templates and modes

```
^^+message
```

Errors

Predicate is a variable:
```
instantiation_error
```
Predicate is declared private:
```
permission_error(access, private_predicate, Predicate)
```
Predicate is not declared:
```
existence_error(predicate_declaration, Predicate)
```
Container of the inherited predicate definition is the same object that contains the `^^/1` call:
```
endless_loop(Predicate)
```

Examples

```
init :-
    assertz(counter(0)),
    ^^init.
```

B.4.2 Calling external code

{}/1

Description

```
{Goal}
```

This control construct enables calling of external Prolog code. It can be used to bypass the Logtalk preprocessor/compiler.

Template and modes

```
{+callable}
```

Errors

(none)

Examples

```
N1/D1 < N2/D2 :-
    {N1*D2 < N2*D1}.
```

Appendix C

Logtalk XML documenting files

As described in Chapter 7, the Logtalk compiler outputs a documenting file in XML format whenever an entity is compiled. This appendix formally defines the XML documenting file format. In addition, it includes two examples on how to convert a XML file to a more human-readable format such as HTML and PDF using XSLT.

C.1 Logtalk XML documenting files structure

Compiling a Logtalk entity outputs an XML [104] documentation file whose structure is here formally defined using both Document Type Definition (DTD) [105] and Schema [106] syntaxes. The corresponding `logtalk.dtd` and `logtalk.xsd` files are usually used by XML and XSLT [107] parsers when validating and processing Logtalk XML files.

C.1.1 Logtalk XML DTD

A DTD file is the most common way of describing the structure of a XML file. This format is supported by most XML and XSLT processors. However, it is a rather limited format for describing the contents of elements, in particular if we want some kind of data typing to express a range for an element contents or a list of valid content values.

```
<!ELEMENT logtalk (entity, relations, predicates)>

<!ELEMENT entity
  (name, type, compilation,
   comment?, author?, version?, date?, info*)>

<!ELEMENT name (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT compilation (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT info (key, value)>

<!ELEMENT key (#PCDATA)>
```

```
<!ELEMENT value (#PCDATA)>

<!ELEMENT relations
  (implements*, imports*, extends*, instantiates*, specializes*,
   uses*, calls*)>

<!ELEMENT implements (name, scope, file)>
<!ELEMENT imports (name, scope, file)>
<!ELEMENT extends (name, scope, file)>
<!ELEMENT instantiates (name, scope, file)>
<!ELEMENT specializes (name, scope, file)>

<!ELEMENT uses (name, file)>
<!ELEMENT calls (name, file)>

<!ELEMENT scope (#PCDATA)>
<!ELEMENT file (#PCDATA)>

<!ELEMENT predicates (public, protected, private)>

<!ELEMENT public (predicate*)>
<!ELEMENT protected (predicate*)>
<!ELEMENT private (predicate*)>

<!ELEMENT predicate
  (name, scope, compilation,
   meta?, mode*, comment?, template?, info*)>

<!ELEMENT meta (#PCDATA)>

<!ELEMENT mode (template, solutions)>

<!ELEMENT template (#PCDATA)>
<!ELEMENT solutions (#PCDATA)>
```

C.1.2 Logtalk XML Schema

The XML Schema language is an emerging standard for describing both the structure and the contents of XML files. Unlike the DTD format, a XML Schema is expressed in XML, providing a much more expressive description language, which enables us to document not only the structure but also the contents of an XML document. We take advantage of this expressive power to define most element contents types and valid values.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:annotation>
  <xsd:documentation>
```

```
  XML Schema for Logtalk XML documentation files.
  </xsd:documentation>
</xsd:annotation>

<xsd:element name="logtalk" type="logtalk"/>

<xsd:complexType name="logtalk">
  <xsd:sequence>
    <xsd:element name="entity"
           type="entity"/>
    <xsd:element name="relations"
           type="relations"/>
    <xsd:element name="predicates"
           type="predicates"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="entity">
  <xsd:sequence>
    <xsd:element name="name"
           type="xsd:string"/>
    <xsd:element name="type"
           type="type"/>
    <xsd:element name="compilation"
           type="compilation"/>
    <xsd:element name="comment"
           type="xsd:string"
           minOccurs="0"/>
    <xsd:element name="author"
           type="xsd:string"
           minOccurs="0"/>
    <xsd:element name="version"
           type="xsd:string"
           minOccurs="0"/>
    <xsd:element name="date"
           type="xsd:string"
           minOccurs="0"/>
    <xsd:element name="info"
           type="info"
           minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="type">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="category"/>
    <xsd:enumeration value="object"/>
    <xsd:enumeration value="protocol"/>
```

```
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="compilation">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="static"/>
    <xsd:enumeration value="dynamic"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="info">
  <xsd:sequence>
    <xsd:element name="key"
           type="xsd:string"/>
    <xsd:element name="value"
           type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="relations">
  <xsd:sequence>
    <xsd:element name="implements"
           type="relation"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="imports"
           type="relation"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="extends"
           type="relation"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="instantiates"
           type="relation"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="specializes"
           type="relation"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="uses"
           type="docrelation"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="calls"
           type="docrelation"
           minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="relation">
  <xsd:sequence>
    <xsd:element name="name"
```

```
            type="xsd:string"/>
    <xsd:element name="scope"
            type="scope"/>
    <xsd:element name="file"
            type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="docrelation">
  <xsd:sequence>
    <xsd:element name="name"
            type="xsd:string"/>
    <xsd:element name="file"
            type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="scope">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="public"/>
    <xsd:enumeration value="protected"/>
    <xsd:enumeration value="private"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="predicates">
  <xsd:sequence>
    <xsd:element name="public"
            type="public"/>
    <xsd:element name="protected"
            type="protected"/>
    <xsd:element name="private"
            type="private"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="public">
  <xsd:sequence>
    <xsd:element name="predicate"
            type="predicate"
            minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="protected">
  <xsd:sequence>
    <xsd:element name="predicate"
            type="predicate"
```

```
            minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="private">
  <xsd:sequence>
    <xsd:element name="predicate"
            type="predicate"
            minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="predicate">
  <xsd:sequence>
    <xsd:element name="name"
            type="xsd:string"/>
    <xsd:element name="scope"
            type="scope"/>
    <xsd:element name="compilation"
            type="compilation"/>
    <xsd:element name="meta"
            type="xsd:string"
            minOccurs="0"/>
    <xsd:element name="mode"
            type="mode"
            minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="comment"
            type="xsd:string"
            minOccurs="0"/>
    <xsd:element name="template"
            type="xsd:string"
            minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="info"
            type="info"
            minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="mode">
  <xsd:sequence>
    <xsd:element name="template"
            type="xsd:string"/>
    <xsd:element name="solutions"
            type="solutions"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="solutions">
```

```
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="zero"/>
    <xsd:enumeration value="zero_or_one"/>
    <xsd:enumeration value="zero_or_more"/>
    <xsd:enumeration value="one"/>
    <xsd:enumeration value="one_or_more"/>
    <xsd:enumeration value="error"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

C.2 Example Logtalk XML documenting file

This section contains an example of a Logtalk automatically generated XML documenting file for a protocol named `listp`. This protocol contains declarations for three common list predicates: `append/3`, `length/2`, and `member/2`. Here is the listing of the `listp` protocol code:

```
:- protocol(listp).

    :- info([
        version is 1.0,
        author is 'Paulo Moura',
        date is 2000/7/24,
        comment is 'List protocol.']).

    :- public(append/3).
    :- mode(append(?list, ?list, ?list), zero_or_more).
    :- info(append/3, [
        comment is 'Appends two lists.',
        argnames is ['List1', 'List2', 'List']]).

    :- public(length/2).
    :- mode(length(?list, ?integer), zero_or_more).
    :- info(length/2, [
        comment is 'List length.',
        argnames is ['List', 'Length']]).

    :- public(member/2).
    :- mode(member(?term, ?list), zero_or_more).
    :- info(member/2, [
        comment is 'Element is a list member.',
        argnames is ['Element', 'List']]).

:- end_protocol.
```

Compiling this protocol generates an XML file that contains a reference to an XSLT file (`logtalk.xsl` in this case) which will be used to transform the XML file in another

format (please see the next section). The name of the XSLT file is a compilation option of the Logtalk compiler. Here is a listing of the XML documenting file:

```
<?xml version="1.0"?>
<!DOCTYPE logtalk SYSTEM "logtalk.dtd">
<?xml-stylesheet type="text/xsl" href="logtalk.xsl"?>

<logtalk>
  <entity>
    <name><![CDATA[listp]]></name>
    <type>protocol</type>
    <compilation>static</compilation>
    <comment><![CDATA[List protocol.]]></comment>
    <author><![CDATA[Paulo Moura]]></author>
    <version>1.0</version>
    <date>2000/7/24</date>
  </entity>
  <relations>
  </relations>
  <predicates>
    <public>
      <predicate>
        <name><![CDATA[append/3]]></name>
        <scope>public</scope>
        <compilation>static</compilation>
        <mode>
          <template><![CDATA[append(?list, ?list, ?list)]]></template>
          <solutions>zero_or_more</solutions>
        </mode>
        <comment><![CDATA[Appends two lists.]]></comment>
        <template><![CDATA[append(List1, List2, List)]]></template>
      </predicate>
      <predicate>
        <name><![CDATA[length/2]]></name>
        <scope>public</scope>
        <compilation>static</compilation>
        <mode>
          <template><![CDATA[length(?list, ?integer)]]></template>
          <solutions>zero_or_more</solutions>
        </mode>
        <comment><![CDATA[List length.]]></comment>
        <template><![CDATA[length(List, Length)]]></template>
      </predicate>
      <predicate>
        <name><![CDATA[member/2]]></name>
        <scope>public</scope>
        <compilation>static</compilation>
        <mode>
```

```
          <template><![CDATA[member(?term, ?list)]]></template>
          <solutions>zero_or_more</solutions>
        </mode>
        <comment><![CDATA[Element is a list member.]]></comment>
        <template><![CDATA[member(Element, List)]]></template>
      </predicate>
    </public>
    <protected>
    </protected>
    <private>
    </private>
  </predicates>
</logtalk>
```

Note the use of `CDATA` tags to avoid problems with user-defined text (such as predicate names and comments) that may contain characters which otherwise would need to be escaped to be in conformance with the XML syntax.

C.3 Example XSLT processing files

This section contains two examples of XSLT files that will transform a Logtalk XML documenting file into nicely formatted HTML and PDF pages. These files are typically used in shell scripts that automate the conversion and indexing of sets of documenting files.

C.3.1 Converting documenting files to HTML

The XSLT file described here as an example, converts an XML documenting file into nicely formatted, standards compliant, HTML file. If the documented entity contains references to other entities, for example, inheritance relationships, the HTML file will include links to the pages documenting the related entities. The HTML page also contains a reference to a CSS [118] file that has not been included here. This CSS file is used to define the final appearance of the HTML file.

```
<?xml version="1.0"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
  method="html"
  doctype-public="-//W3C//DTD HTML 4.01//EN"
  doctype-system="http://www.w3.org/TR/html4/strict.dtd"/>

<xsl:template match="/">
  <html>
  <head>
    <title><xsl:value-of select="logtalk/entity/name" /></title>
    <link rel="stylesheet" href="logtalk.css" type="text/css" />
```

```
  </head>
  <body>
    <hr />
    <h4 class="type">
      <xsl:value-of select="logtalk/entity/type" />
    </h4>
    <h1 class="code">
      <xsl:value-of select="logtalk/entity/name" />
    </h1>
    <xsl:apply-templates select="logtalk/entity" />
    <hr />
    <xsl:apply-templates select="logtalk/relations" />
    <hr />
    <xsl:apply-templates select="logtalk/predicates" />
    <hr />
  </body>
  </html>
</xsl:template>

<xsl:template match="logtalk/entity">
  <xsl:if test="comment">
    <blockquote>
      <p class="blockquote"><xsl:value-of select="comment" /></p>
    </blockquote>
  </xsl:if>
  <dl>
  <xsl:if test="author">
    <dt>author:</dt>
      <dd><code><xsl:value-of select="author" /></code></dd>
  </xsl:if>
  <xsl:if test="version">
    <dt>version:</dt>
      <dd><code><xsl:value-of select="version" /></code></dd>
  </xsl:if>
  <xsl:if test="date">
    <dt>date:</dt>
      <dd><code><xsl:value-of select="date" /></code></dd>
  </xsl:if>
  </dl>
  <dl>
    <dt>compilation:</dt>
      <dd><code><xsl:value-of select="compilation" /></code></dd>
  </dl>
  <xsl:if test="info">
    <dl>
    <xsl:for-each select="info">
      <dt><xsl:value-of select="key" />:</dt>
        <dd><code><xsl:value-of select="value" /></code></dd>
```

```
    </xsl:for-each>
    </dl>
  </xsl:if>
</xsl:template>

<xsl:template match="logtalk/relations">
  <xsl:choose>
    <xsl:when test="*">
      <xsl:if test="implements">
      <dl>
        <dt>implements:</dt>
        <xsl:apply-templates select="implements" />
      </dl>
      </xsl:if>
      <xsl:if test="imports">
      <dl>
        <dt>imports:</dt>
        <xsl:apply-templates select="imports" />
      </dl>
      </xsl:if>
      <xsl:if test="extends">
      <dl>
        <dt>extends:</dt>
        <xsl:apply-templates select="extends" />
      </dl>
      </xsl:if>
      <xsl:if test="instantiates">
      <dl>
        <dt>instantiates:</dt>
        <xsl:apply-templates select="instantiates" />
      </dl>
      </xsl:if>
      <xsl:if test="specializes">
      <dl>
        <dt>specializes:</dt>
        <xsl:apply-templates select="specializes" />
      </dl>
      </xsl:if>
      <xsl:if test="uses">
      <dl>
        <dt>uses:</dt>
        <xsl:apply-templates select="uses" />
      </dl>
      </xsl:if>
      <xsl:if test="calls">
      <dl>
        <dt>calls:</dt>
        <xsl:apply-templates select="calls" />
```

```
      </dl>
      </xsl:if>
    </xsl:when>
    <xsl:otherwise>
      <h4 class="code">(no dependencies on other files)</h4>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="logtalk/relations/uses">
  <dd><code>
    <a href="{file}.html"><xsl:value-of select="name" /></a>
  </code></dd>
</xsl:template>

<xsl:template match="logtalk/relations/calls">
  <dd><code>
    <a href="{file}.html"><xsl:value-of select="name" /></a>
  </code></dd>
</xsl:template>

<xsl:template match="logtalk/relations/*">
  <dd><code>
    <xsl:value-of select="scope" /><xsl:text> </xsl:text>
    <a href="{file}.html"><xsl:value-of select="name" /></a>
  </code></dd>
</xsl:template>

<xsl:template match="logtalk/predicates">
  <h1>Public interface</h1>
  <xsl:choose>
    <xsl:when test="public/predicate">
      <xsl:apply-templates select="public/predicate" />
    </xsl:when>
    <xsl:when test="/logtalk/relations/*">
      <h4 class="code">(see related entities)</h4>
    </xsl:when>
    <xsl:otherwise>
      <h4 class="code">(none)</h4>
    </xsl:otherwise>
  </xsl:choose>
  <h1>Protected interface</h1>
  <xsl:choose>
    <xsl:when test="protected/predicate">
      <xsl:apply-templates select="protected/predicate" />
    </xsl:when>
    <xsl:when test="/logtalk/relations/*">
      <h4 class="code">(see related entities)</h4>
```

```
    </xsl:when>
    <xsl:otherwise>
      <h4 class="code">(none)</h4>
    </xsl:otherwise>
  </xsl:choose>
  <h1>Private predicates</h1>
  <xsl:choose>
    <xsl:when test="private/predicate">
      <xsl:apply-templates select="private/predicate" />
    </xsl:when>
    <xsl:when test="/logtalk/relations/*">
      <h4 class="code">(see related entities)</h4>
    </xsl:when>
    <xsl:otherwise>
      <h4 class="code">(none)</h4>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="*/predicate">
  <h4 class="code"><xsl:value-of select="name" /></h4>
  <xsl:if test="comment">
    <blockquote><p class="blockquote">
      <xsl:value-of select="comment" />
    </p></blockquote>
  </xsl:if>
  <dl class="predicate">
    <dt>compilation:</dt>
      <dd><code><xsl:value-of select="compilation" /></code></dd>
    <xsl:if test="template">
    <dt>template:</dt>
      <dd><code><xsl:value-of select="template" /></code></dd>
    </xsl:if>
    <xsl:if test="meta">
    <dt>metapredicate template:</dt>
      <dd><code><xsl:value-of select="meta" /></code></dd>
    </xsl:if>
    <xsl:if test="mode">
    <dt>mode - number of solutions:</dt>
    <xsl:for-each select="mode">
      <dd><code>
        <xsl:value-of select="template" />
        <xsl:text> - </xsl:text>
        <xsl:value-of select="solutions" />
      </code></dd>
    </xsl:for-each>
    </xsl:if>
  </dl>
```

```
  <xsl:if test="info">
    <dl class="predicate">
      <xsl:for-each select="info">
      <dt><xsl:value-of select="key" />:</dt>
        <dd><code><xsl:value-of select="value" /></code></dd>
      </xsl:for-each>
    </dl>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

C.3.2 Converting documenting files to PDF

We can also convert the Logtalk XML documenting files to a ready to print format such as PDF using XSL Formatting Objects (XSL-FO) [107]. The XSL-FO standard is currently a work-in-progress. Some of the applications supporting it are Apache's FOP processor [119], PassiveTeX TEX macros [120], and RenderX's XEP [121]. Here is an example stylesheet for converting a documenting file to a PDF file formatted for an A4 paper printer:

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:output indent="yes"/>

<xsl:template match ="/">

  <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

    <fo:layout-master-set>
      <fo:simple-page-master
          master-name="simple"
          page-height="297mm"
          page-width="210mm"
          margin-top="15mm"
          margin-bottom="15mm"
          margin-left="25mm"
          margin-right="25mm">
        <fo:region-body margin-top="15mm" margin-bottom="15mm"/>
        <fo:region-before extent="15mm"/>
        <fo:region-after extent="15mm"/>
      </fo:simple-page-master>
    </fo:layout-master-set>

    <fo:page-sequence master-reference="simple">
```

```
      <fo:static-content flow-name="xsl-region-before">
        <fo:block>
          <fo:leader leader-pattern="rule" leader-length="100%"/>
        </fo:block>
        <fo:block
            text-align="end"
            font-size="9pt"
            font-family="sans-serif"
            font-weight="bold">
          <xsl:value-of select="logtalk/entity/type"/>
            <xsl:text>: </xsl:text>
          <xsl:value-of select="logtalk/entity/name"/>
        </fo:block>
      </fo:static-content>

      <fo:static-content flow-name="xsl-region-after">
        <fo:block>
          <fo:leader leader-pattern="rule" leader-length="100%"/>
        </fo:block>
        <fo:block
            text-align="end"
            font-size="9pt"
            font-family="sans-serif"
            font-weight="bold">
          <fo:page-number/> of <fo:page-number-citation ref-id="end"/>
        </fo:block>
      </fo:static-content>

      <fo:flow flow-name="xsl-region-body">
            <fo:block
                font-size="18pt"
                    font-family="sans-serif"
                    font-weight="bold"
                    space-after="8pt">
              <xsl:value-of select="logtalk/entity/name"/>
            </fo:block>
        <xsl:apply-templates select="logtalk/entity"/>
        <xsl:apply-templates select="logtalk/relations"/>
        <xsl:apply-templates select="logtalk/predicates"/>
        <fo:block id="end"/>
      </fo:flow>

    </fo:page-sequence>

  </fo:root>

</xsl:template>
```

```
<xsl:template match="logtalk/entity">

  <xsl:if test="comment">
    <fo:block
        margin-left="10mm"
        font-size="10pt"
        font-family="serif"
        font-style="italic"
        space-after="8pt">
      <xsl:value-of select="comment"/>
    </fo:block>
  </xsl:if>

  <xsl:if test="author">
    <fo:block
        font-size="10pt"
        font-family="serif"
        keep-with-next="always">
      author:
    </fo:block>
    <fo:block
        font-size="9pt"
        font-family="monospace"
        margin-left="10mm">
      <xsl:value-of select="author"/>
    </fo:block>
  </xsl:if>

  <xsl:if test="version">
    <fo:block
        font-size="10pt"
        font-family="serif"
        keep-with-next="always">
      version:
    </fo:block>
    <fo:block
        font-size="9pt"
        font-family="monospace"
        margin-left="10mm">
      <xsl:value-of select="version"/>
    </fo:block>
  </xsl:if>

  <xsl:if test="date">
    <fo:block
        font-size="10pt"
        font-family="serif"
        keep-with-next="always">
```

```
      date:
    </fo:block>
    <fo:block
        font-size="9pt"
        font-family="monospace"
        margin-left="10mm">
      <xsl:value-of select="date"/>
    </fo:block>
  </xsl:if>

  <fo:block
      font-size="10pt"
      font-family="serif"
      space-before="8pt"
      keep-with-next="always">
    compilation:
  </fo:block>
  <fo:block
      font-size="9pt"
      font-family="monospace"
      margin-left="10mm"
      space-after="8pt">
    <xsl:value-of select="compilation"/>
  </fo:block>

  <xsl:if test="info">
    <xsl:for-each select="info">
          <fo:block
          font-size="10pt"
          font-family="serif"
          keep-with-next="always">
          <xsl:value-of select="key"/>:
        </fo:block>
      <fo:block
          font-size="9pt"
          font-family="monospace"
          margin-left="10mm">
        <xsl:value-of select="value"/>
      </fo:block>
    </xsl:for-each>
  </xsl:if>

</xsl:template>

<xsl:template match="logtalk/relations">
  <xsl:choose>
    <xsl:when test="*">
      <xsl:if test="implements">
```

```
  <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
    implements:
  </fo:block>
  <xsl:apply-templates select="implements"/>
</xsl:if>
<xsl:if test="imports">
  <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
    imports:
  </fo:block>
  <xsl:apply-templates select="imports"/>
</xsl:if>
<xsl:if test="extends">
  <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
    extends:
  </fo:block>
  <xsl:apply-templates select="extends"/>
</xsl:if>
<xsl:if test="instantiates">
  <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
    instantiates:
  </fo:block>
  <xsl:apply-templates select="instantiates"/>
</xsl:if>
<xsl:if test="specializes">
  <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
    specializes:
  </fo:block>
  <xsl:apply-templates select="specializes"/>
</xsl:if>
<xsl:if test="uses">
  <fo:block
      font-size="10pt"
      font-family="serif"
```

```
            keep-with-next="always">
          uses:
        </fo:block>
        <xsl:apply-templates select="uses"/>
      </xsl:if>
      <xsl:if test="calls">
        <fo:block
            font-size="10pt"
            font-family="serif"
            keep-with-next="always">
          calls:
        </fo:block>
        <xsl:apply-templates select="calls"/>
      </xsl:if>
    </xsl:when>
    <xsl:otherwise>
      <fo:block
          font-size="10pt"
          font-family="serif"
          keep-with-next="always">
        (no dependencies on other files)
      </fo:block>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="logtalk/relations/uses">
  <fo:block
      font-size="9pt"
      font-family="monospace"
      margin-left="10mm">
    <xsl:value-of select="name"/>
  </fo:block>
</xsl:template>

<xsl:template match="logtalk/relations/calls">
  <fo:block
      font-size="9pt"
      font-family="monospace"
      margin-left="10mm">
    <xsl:value-of select="name"/>
  </fo:block>
</xsl:template>

<xsl:template match="logtalk/relations/*">
  <fo:block
      font-size="9pt"
      font-family="monospace"
```

```
      margin-left="10mm">
    <xsl:value-of select="scope"/>
      <xsl:text> </xsl:text>
    <xsl:value-of select="name"/>
  </fo:block>
</xsl:template>

<xsl:template match="logtalk/predicates">

  <fo:block
      font-size="14pt"
      font-family="sans-serif"
      font-weight="bold"
      keep-with-next="always"
      space-before="18pt">
    Public interface
  </fo:block>
  <xsl:choose>
    <xsl:when test="public/predicate">
      <xsl:apply-templates select="public/predicate"/>
    </xsl:when>
    <xsl:when test="/logtalk/relations/*">
      <fo:block
          font-size="10pt"
          font-family="serif"
          font-style="italic"
          space-before="10pt">
        (see related entities)
      </fo:block>
    </xsl:when>
    <xsl:otherwise>
      <fo:block
          font-size="10pt"
          font-family="serif"
          font-style="italic"
          space-before="10pt">
        (none)
      </fo:block>
    </xsl:otherwise>
  </xsl:choose>

  <fo:block
      font-size="14pt"
      font-family="sans-serif"
      font-weight="bold"
      keep-with-next="always"
      space-before="18pt">
    Protected interface
```

```
</fo:block>
<xsl:choose>
  <xsl:when test="protected/predicate">
    <xsl:apply-templates select="protected/predicate"/>
  </xsl:when>
  <xsl:when test="/logtalk/relations/*">
    <fo:block
        font-size="10pt"
        font-family="serif"
        font-style="italic"
        space-before="10pt">
      (see related entities)
    </fo:block>
  </xsl:when>
  <xsl:otherwise>
    <fo:block
        font-size="10pt"
        font-family="serif"
        font-style="italic"
        space-before="10pt">
      (none)
    </fo:block>
  </xsl:otherwise>
</xsl:choose>

<fo:block
    font-size="14pt"
    font-family="sans-serif"
    font-weight="bold"
    keep-with-next="always"
    space-before="18pt">
  Private predicates
</fo:block>
<xsl:choose>
  <xsl:when test="private/predicate">
    <xsl:apply-templates select="private/predicate"/>
  </xsl:when>
  <xsl:when test="/logtalk/relations/*">
    <fo:block
        font-size="10pt"
        font-family="serif"
        font-style="italic"
        space-before="10pt">
      (see related entities)
    </fo:block>
  </xsl:when>
  <xsl:otherwise>
    <fo:block
```

```
          font-size="10pt"
          font-family="serif"
          font-style="italic"
          space-before="10pt">
        (none)
      </fo:block>
    </xsl:otherwise>
  </xsl:choose>

</xsl:template>

<xsl:template match="*/predicate">

  <fo:block
      font-size="12pt"
      font-family="sans-serif"
      font-weight="bold"
      keep-with-next="always"
      space-before="10pt">
    <xsl:value-of select="name"/>
  </fo:block>

  <xsl:if test="comment">
    <fo:block
        margin-left="10mm"
        font-size="10pt"
        font-family="serif"
        font-style="italic"
        space-before="4pt"
        space-after="8pt">
      <xsl:value-of select="comment"/>
    </fo:block>
  </xsl:if>

  <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
    compilation:
  </fo:block>
  <fo:block
      font-size="9pt"
      font-family="monospace"
      margin-left="10mm">
    <xsl:value-of select="compilation"/>
  </fo:block>

  <xsl:if test="template">
```

```
      <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
      template:
    </fo:block>
  <fo:block
      font-size="9pt"
      font-family="monospace"
      margin-left="10mm">
    <xsl:value-of select="template"/>
  </fo:block>
</xsl:if>

<xsl:if test="meta">
      <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
      metapredicate template:
    </fo:block>
  <fo:block
      font-size="9pt"
      font-family="monospace"
      margin-left="10mm">
    <xsl:value-of select="meta"/>
  </fo:block>
</xsl:if>

<xsl:if test="mode">
      <fo:block
      font-size="10pt"
      font-family="serif"
      keep-with-next="always">
      mode - number of solutions:
    </fo:block>
  <xsl:for-each select="mode">
    <fo:block
        font-size="9pt"
        font-family="monospace"
        margin-left="10mm">
      <xsl:value-of select="template"/>
        <xsl:text> - </xsl:text>
      <xsl:value-of select="solutions"/>
    </fo:block>
  </xsl:for-each>
</xsl:if>
```

```
  <xsl:if test="info">
    <xsl:for-each select="info">
          <fo:block
          font-size="10pt"
          font-family="serif"
          keep-with-next="always">
          <xsl:value-of select="key"/>:
        </fo:block>
      <fo:block
          font-size="9pt"
          font-family="monospace"
          margin-left="10mm">
        <xsl:value-of select="value"/>
      </fo:block>
    </xsl:for-each>
  </xsl:if>

</xsl:template>

</xsl:stylesheet>
```

Bibliography

[1] Alain Colmerauer and Philippe Roussel. The birth of Prolog, November 1992.

[2] ISO/IEC. *International Standard ISO/IEC 13211-1 Information Technology — Programming Languages — Prolog — Part I: General core.* ISO/IEC, 1995.

[3] ISO/IEC. *International Standard ISO/IEC 13211-2 Information Technology — Programming Languages — Prolog — Part II: Modules.* ISO/IEC, 2000.

[4] Asian Technology Information Program. ATIP98.095: Current FGCS Technology and Activities in Japan. `http://www.atip.org/public/atip.reports.98/atip98.095.html/`, November 1998.

[5] ACM/IEEE. Computer Curricula 2001: Volume II — Computer Science — Strawman Draft. `http://www.computer.org/education/cc2001/`, March 2000.

[6] ACM/IEEE. Computer Curricula 2001 — Computer Science — Final Report. `http://www.computer.org/education/cc2001/`, December 2001.

[7] The Computational Logic, Implementation, and Parallelism Group, UPM, Spain. Logic Programming and the IEEE/ACM 2001 CS Curriculum. `http://www.clip.dia.fi.upm.es/logic_programming_curr/`, July 2000.

[8] Bjarne Stroustrup. *The C++ Programming Language.* Series in Computer Science. Addison-Wesley, 3rd edition, 1997.

[9] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[10] James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-Based Programming - Concepts, Languages and Applications.* Springer-Verlag, 1999.

[11] James E. Rumbaugh. Relations as semantic constructs in an object-oriented language. In Meyrowitz [122], pages 466–481.

[12] James E. Rumbaugh. Controlling propagation of operations using attributes on relations. In Meyrowitz [123], pages 285–296.

[13] M. Fornarino, A.-M. Pinna, and B. Trousse. An original object-oriented approach for relation management. In J. P. Martins and E. M. Morgado, editors, *4th Portuguese Conference on Artificial Intelligence*, volume 390 of *Lecture Notes in Artificial Intelligence*, pages 13–26. Springer-Verlag, September 1989.

[14] Gottfried Razek. Combining objects and relations. *Comunications of the ACM*, 12(27):66–70, 1992.

[15] Pattie Maes. Concepts and experiments in computational reflection. In Meyrowitz [122], pages 147–155.

[16] A. Tanenbaum. *Operating Systems — Design and Implementation*. Software Series. Prentice-Hall, 1987.

[17] J. Kunz, T. P. Kehler, and M. D. Williams. Applications development using a hybrid ai development system. *AI Magazine*, 5(4), 1984.

[18] Mark J. Stefik, Daniel G. Bobrow, and Ken M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, pages 10–18, January 1986.

[19] Adele Goldberg and David Robson. *Smalltalk-80 — The language and its implementation*. Series in Computer Science. Addison-Wesley, 1983.

[20] Henry Lieberman. Using prototypical objects to implement shared behaviour in object oriented systems. In Meyrowitz [124], pages 189–214.

[21] Brad J. Cox and Andrew Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 2nd edition, June 1991.

[22] David Taenzer, Murhty Ganti, and Sunil Podar. Problems in object-oriented software reuse. In S. Cook, editor, *ECOOP 89, European Conference on Object-Oriented Programming*, British Computer Society Workshop Series, pages 25–38. Cambridge University Press, July 1989.

[23] Francis G. McCabe. *Logic and Objects*. Series in Computer Science. Prentice Hall, 1992.

[24] Swedish Institute for Computer Science. SICStus Prolog Home Page. `http://www.sics.se/isl/sicstus/`.

[25] Paulo Moura. Logtalk 2.6 Documentation. Technical Report DMI 2000/1, University of Beira Interior, Portugal, 2000.

[26] Paulo Moura. Porting Prolog: Notes on porting a Prolog program to 22 Prolog compilers or the relevance of the ISO Prolog standard. *Assotiation of Logic Programming Newsletter*, 12(2), May 1999.

[27] Paulo Moura. Logtalk: Programação Orientada em Objectos em Prolog. In A. V. Velho, editor, *Segunda Conferência e Exposição Portuguesa de Tecnologia Orientada por Objectos*, pages 234–239. 3i Consultores, Lisboa, September 1994.

[28] Paulo Moura. Logtalk web site. `http://www.logtalk.org/`.

[29] Antero Taivalsaari. *Prototype-Based Programming: Concepts, Languages and Applications*, chapter Classes vs. Prototypes: Some Philosophical and Historical Observations, pages 3–16. Springer Verlag, 1999.

[30] Jacques Malenfant. Object-centered programming. In *European Conference on Object-Oriented Programming, Workshop on Prototype-Based Object-Oriented Programming*, July 1996.

[31] Henry Lieberman, Lynn Andrea Stein, and David Ungar. *Object-Oriented Concepts, Applications and Databases*, chapter The Treaty of Orlando: A Shared View of Sharing. Addison-Wesley, 1988.

[32] David Ungar, Henry Lieberman, Lynn Andrea Stein, and Daniel Halbert. Panel: Treaty of orlando revisited. In Meyrowitz [123].

[33] Alan Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 36–40, Dallas, Texas, November 1986.

[34] Paulo Moura. Logtalk: Programação Orientada em Objectos em Prolog. Master's thesis, Departamento de Engenharia Informática, Universidade de Coimbra, Portugal, December 1995.

[35] Peter Schachte and Georges Saab. Efficient object-oriented programming in prolog. In *Logic Programming: Formal Methods and Pratical Applications*, number 11 in Studies in Computer Science and Artificial Intelligence. Elsevier Science B.V. North-Holland, Amsterdam, 1995.

[36] Swedish Institute for Computer Science. Quintus Prolog Home Page. `http://www.sics.se/isl/quintus/`.

[37] Swedish Institute for Computer Science. *SICStus Prolog 3.10.1 User Manual*. Swedish Institute for Computer Science, April 2003.

[38] BinNet Corporation. Jinni — Java and Prolog software for Internet programming. `http://www.binnetcorp.com/Jinni/`.

[39] BinNet Corporation. Jinni 2003 Prolog Compiler - User Guide. `http://www.binnetcorp.com/download/jinnidemo/JinniUserGuide.html`.

[40] Eric Borgers. OPL User Manual. `http://www.amzi.com/download/freedist_opl.htm`, 2001.

[41] Amzi! Inc. Amzi! prolog. `http://www.amzi.com/`.

[42] Angel Fernandez Pineda and Francisco Bueno. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint and Logic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[43] Angel Fernandez Pineda and Manuel Hermenegildo. O'Ciao — An Object Oriented Programming model using CIAO Prolog. Technical Report CLIP 5/99.0, The CLIP Group, School of Computer Science, Technical University of Madrid, July 1999.

[44] Francisco Bueno, Daniel Cabeza, Manuel Carro, Manuel Hermenegildo, Pedro López, and Germán Puebla. The Ciao Prolog System. Technical Report CLIP 3/97.1, The CLIP Group, School of Computer Science, Technical University of Madrid, December 2002.

[45] Computational Logic, Implementation, and Parallelism Lab. The Ciao Prolog Development System WWW Site. `http://www.clip.dia.fi.upm.es/Software/Ciao/index.html`.

[46] IF Computer GmbH. Minerva: Prolog in java. `http://www.ifcomputer.com/MINERVA/`.

[47] Chris Moss. *Prolog++ The Power of Object-Oriented and Logic Programming*. Series in Logic Programming. Addison-Wesley, 1994.

[48] Dave Westwood. *Prolog++ Reference*, 1995.

[49] Logic Programming Associates Ltd. LPA Home Page. `http://www.lpa.co.uk/`.

[50] Salvador Abreu. ISCO: A practical language for heterogeneous information system construction. In *Proceedings of 14th International Conference of Applications of Prolog*, pages 107–119, Tokyo, Japan, October 2001.

[51] Anjo Anjewierden and Jan Wielemaker. Xpce: the swi-prolog native gui library. `http://www.swi-prolog.org/packages/xpce/`.

[52] Markus Fromherz. OL(P): Object Layer for Prolog. `ftp://parcftp.xerox.com/ftp/pub/ol/`.

[53] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1st edition, 1994.

[54] Dan G. Bobrow, Linda G. Michiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczale, and David A. Moon. Common lisp object system specification. *SIGPLAN Notices*, 23, 1988.

[55] Iain Craig. *The Interpretation of Object-Oriented Programming Languages*. Springer Verlag, December 1999.

[56] Yen-Ping Shan, Thomas A. Cargill, Brad Cox, William Cook, Mary Loomis, and Alan Snyder. Is multiple inheritance essential to OOP? (panel). In Andreas Paepcke, editor, *OOPLSLA 93, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 28 of *SIGPLAN Notices*, pages 360–363. ACM Press, October 1993.

[57] Pierre Cointe. Metaclasses are first class: the ObjVlisp model. In Meyrowitz [122], pages 156–167.

[58] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1987.

[59] Christophe Dony, Jaques Malenfant, and Daniel Bardau. *Prototype-Based Programming: Concepts, Languages and Applications*, chapter Classifying Prototype-based Programming Languages, pages 17–45. Springer Verlag, 1999.

[60] David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp And Symbolic Computation*, 4(3), 1991.

[61] D.L. Bowen, L. Byrd, F. C. N. Pereira, L. M. Pereira, and D. H. D. Warren. *DEC-10 Prolog Users Manual*. Department of Artificial Intelligence, University of Edinburgh, November 1982.

[62] David H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[63] IC-Parc. The ECLiPSe Constraint Logic Programming System. `http://www.icparc.ic.ac.uk/eclipse/`.

[64] The XSB Research Group. XSB Home Page. `http://xsb.sourceforge.net/`.

[65] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The implementation of Mercury: an efficient purely declarative logic programming language. In *Proceedings of the ILPS'94 Postconference Workshop on Implementation Techniques for Logic Programming Languages, Syracuse, New York*, November 1994.

[66] University of Melbourne. The Mercury Project: Introduction. `http://www.cs.mu.oz.au/research/mercury/`.

[67] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. *Mercury Language Reference Manual*. The Mercury Project.

[68] Vitor Santos Costa. YAP Home Page. `http://www.cos.ufrj.br/~vitor/Yap/`.

[69] Jan Wielemaker. SWI-Prolog Home Page. `http://www.swi-prolog.org/`.

[70] BinNet Corporation. BinProlog Home Page. `http://www.binnetcorp.com/BinProlog/`.

[71] Swedish Institute for Computer Science. SICStus Prolog Manual. `http://www.sics.se/isl/sicstus.html`.

[72] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[73] Jonathan Hodgson. ISO/IEC/ JTC1/SC22/WG17 official home page. `http://www.sju.edu/~jhodgson/wg17/wg17web.html`.

[74] Anthony Dodd. Definite Clause Grammars (DCGs) in ISO Prolog — a proposal. `http://www.sju.edu/~jhodgson/wg17/d895.ps`, September 1992.

[75] ObjectShare, Inc. *VisualWorks Application Developer's Guide, VisualWorks Software Release 3.0*, 1998.

[76] ObjectShare, Inc. Objectshare web site. `http://www.objectshare.com/`.

[77] Apple Computer, Inc. The Objective-C language. `http://developer.apple.com/techpubs/index.html`, May 2002.

[78] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future – the story of Squeak, a practical Smalltalk written in itself. In *OOPLSLA 97, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 21 of *SIGPLAN Notices*, pages 318–326. ACM Press, October 1997.

[79] Squeak.org. Squeak home page. `http://www.squeak.org/`.

[80] Sun Microsystems, Inc. Javasoft web site. `http://www.javasoft.com/`.

[81] Apple Computer, Inc. *Apple Computer Technical Documentation: MacOS X Server — Foundation Framework Classes*, 1999.

[82] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In Andreas Paepcke, editor, *OOPLSLA 92, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 27 of *SIGPLAN Notices*, pages 1–15. ACM Press, October 1992.

[83] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, May 1999.

[84] David A. Moon. Object-oriented programming in Flavors. In Meyrowitz [124], pages 1–8.

[85] Guy L. Steele. *Common LISP: The Language*. Digital Press, Bedford, Massachusetts, 1984.

[86] Gilad Bracha and William Cock. Mixin-based inheritance. In Norman Meyrowitz, editor, *OOPLSLA 90, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 25 of *SIGPLAN Notices*, pages 303–311. ACM Press, October 1990.

[87] Yukihiro Matsumoto. Ruby home page. `http://www.ruby-lang.org/en/`.

[88] David Thomas and Andrew Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley Longman, 2001.

[89] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly, November 2001.

[90] Yukihiro Matsumoto. *Ruby Programmers Reference Guide*, 2000.

[91] Randall B. Smith and David Ungar. Programming as an experience: The inspiration of Self. In W. Olthoff, editor, *ECOOP 95, European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 303–330, Aarhus, Denmark, August 1995. Springer-Verlag.

[92] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP 97, European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[93] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. *Specifying Subject-Oriented Composition, Theory and Practice of Object Systems*, volume 2. Wiley & Sons, 1996.

[94] Ralph Keller and Urs Hölzle. Binary component adaptation. In Eric Jul, editor, *ECOOP 98, European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329, Brussels, Belgium, July 1998. Springer-Verlag.

[95] J. Malenfant, M. Jaques, and F.-N. Demers. A tutorial on behavioral reflection and its implementation. In Gregor Kiczales, editor, *Reflection 96 Conference*, pages 1–20, April 1996.

[96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[97] Sun Microsystems, Inc. Java 2 Platform, Standard Edition, v 1.4.1 API Specification. `http://java.sun.com/j2se/1.4.1/docs/api`.

[98] Donald E. Knuth. The literate programming paradigm. *Computer Journal*, 27(2):97–111, May 1984.

[99] Lisa Friendly. The design of distributed hyperlinked programming documentation. In *International Workshop on Hypermedia Design '95*, 1995.

[100] Computational Logic, Implementation, and Parallelism Lab. The lpdoc Documentation Generator. `http://www.clip.dia.fi.upm.es/Software/Ciao/index.html`.

[101] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*, January 2001.

[102] World Wide Web Consortium. Hypertext Markup Language (html). `http://www.w3.org/MarkUp/`.

[103] LaTeX3 Project. Latex project home page. `http://www.latex-project.org/`.

[104] World Wide Web Consortium. Extensible Markup Language (xml). `http://www.w3.org/XML/`.

[105] Simon St. Laurent and Robert J. Biggar. *Inside XML DTDs: Scientific and Technical*. McGraw-Hill, 1999.

[106] World Wide Web Consortium. W3C XML Schema. `http://www.w3.org/XML/Schema`.

[107] World Wide Web Consortium. Extensible Stylesheet Language (xsl). `http://www.w3.org/Style/XSL/`.

[108] Adobe Systems Incorporated. *PDF Reference — Adobe Portable Document Format Version 1.4*. Addison-Wesley, 3rd edition, 2000.

[109] World Wide Web Consortium. Extensible HyperText Markup Language (XHTML). `http://www.w3.org/MarkUp/`.

[110] OpenSource.Org. Open Source Initiative OSI - The Artistic License:Licensing. `http://www.opensource.org/licenses/artistic-license.php`.

[111] Daniel Diaz. The GNU Prolog web site. `http://gprolog.inria.fr/`.

[112] Pierre Deransart and AbdelAli Ed-Dbali. Executable specification for Standard Prolog version 1.0. `ftp://ftp-lifo.univ-orleans.fr/pub/Users/eddbali/SdProlog/`.

[113] Jonanthan Hodgson. Validation suite of tests for ISO standard conformance. `http://www.sju.edu/~jhodgson/x3j17.html`.

[114] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[115] Open Source Development Network. freshmeat.net. `http://freshmeat.net/`.

[116] British Standards Institute. *British Standard 6154 — Method of defining — Syntactic meta-language*. British Standards Institute, 1981.

[117] Roger Scowen. An Introduction and Handbook for the Standard Syntactic Metalanguage. Technical Report DITC 19/8, National Physical Laboratory, 1983.

[118] World Wide Web Consortium. Cascading Style Sheets (css). `http://www.w3.org/Style/CSS/`.

[119] The Apache XML Project. FOP. `http://xml.apache.org/fop`.

[120] Sebastian Rahtz. PassiveTeX. `http://www.tei-c.org.uk/Software/passivetex/`.

[121] RenderX Corporation. RenderX. `http://www.renderx.com`.

[122] Norman Meyrowitz, editor. *OOPLSLA 87, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 22 of *SIGPLAN Notices*. ACM Press, October 1987.

[123] Norman Meyrowitz, editor. *OOPLSLA 88, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 23 of *SIGPLAN Notices*. ACM Press, November 1988.

[124] Norman Meyrowitz, editor. *OOPLSLA 86, ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 21 of *SIGPLAN Notices*. ACM Press, 1986.

Index