Simone Diniz Junqueira Barbosa

# Programação Via Interface

Tese apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos necessários à obtenção do grau de Doutor em Ciências em Informática

**Orientadora**: Clarisse Sieckenius de Souza

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, 5 de maio de 1999

# Resumo

A indústria de software vem ao longo dos anos aumentando a funcionalidade das aplicações, numa tentativa de satisfazer as necessidades do maior número de usuários possível. Esta solução, no entanto, implica grandes desafios de usabilidade, devido à complexidade cada vez maior destas aplicações. Uma tendência que visa a acomodar as necessidades dos usuários sem sobrecarregar o software com funcionalidade de uso infreqüente é permitir que os próprios usuários finais configurem ou programem as aplicações, através de mecanismos de extensão que suportam um tipo específico de programação, chamado *programação feita por usuários finais*. Entretanto, grande parte das técnicas existentes para tal não conseguem atingir níveis aceitáveis de utilidade e usabilidade.

Este trabalho trata alguns desafios de aplicações extensíveis, propondo uma abordagem que rompe com algumas barreiras entre interface e extensão. Esta abordagem traz para a interface, e ao alcance dos usuários finais, mecanismos de extensão de software com base em recursos semântico–pragmáticos, utilizando cálculos de metáforas e metonímias. Estes mecanismos foram escolhidos devido ao reconhecimento das Ciências Cognitivas do papel que desempenham em nosso raciocínio, em especial quando tentamos descrever ou entender um conceito abstrato ou complexo (Lakoff e Johnson, 1980; Lakoff, 1987; Lakoff, 1993; Ortony, 1993).

Descrevemos um modelo de aplicações extensíveis que utiliza uma base de conhecimento onde devem ser representados os elementos do domínio e da aplicação que podem ser estendidos, bem como as classificações necessárias aos mecanismos de extensão. Nosso modelo considera os aspectos comunicativos das aplicações computacionais. Para garantir a consistência entre a aplicação original e a aplicação estendida, seguimos princípios da Engenharia Semiótica (de Souza, 1993) e prevemos, no modelo, a representação de regras que restringem as extensões na interface, a fim de refletir adequadamente as extensões de funcionalidade.

# Abstract


In the past few years, we have witnessed an increase in software functionality as an attempt to meet most users' needs. This approach brings about serious usability challenges, due to an increase in application complexity as well. In order to try and meet users' needs, without overloading the application with functionality that is rarely used, there is a tendency to allow end users to configure or program applications, by means of mechanisms that support the so-called *end user programming*. However, many existing techniques fail to attain acceptable thresholds of usefulness and usability.

This work addresses some of the challenges posed by extensible applications. We follow an approach that drops some walls between interface and extension. This approach brings some extension mechanisms to the interface, and readily accessible to end-users, namely extensions based on the semantic–pragmatic resources of metaphors and metonymies. These mechanisms were chosen due to the acknowledgment of the Cognitive Sciences of their critical role in our reasoning processes, especially when we try to describe or understand complex or abstract concepts (Lakoff e Johnson, 1980; Lakoff, 1987; Lakoff, 1993; Ortony, 1993).

We describe an extensible application model that makes use of a knowledge base in which we represent the domain and application elements that may be extended, as well as the necessary classifications for calculating the possible extensions. Our model takes into account the communicative aspects of computer applications, and follows Semiotic Engineering (de Souza, 1993) principles to guarantee the consistency between the original application and the extended one. For that purpose, our model entails the representation of rules that constrain interface amendments, so that extended functionality is adequately reflected at the resulting interface.

# Índice Analítico


**Resumo** i

**Abstract** ii

**Índice Analítico** iii

**Índice de Figuras** v

**Índice de Tabelas** vi

**Notação utilizada nesta tese** vii

**Agradecimentos** viii

**1. INTRODUÇÃO** 1

**2. FUNDAMENTOS TEÓRICOS** 7

**2.1 Aplicações extensíveis** 7
- 2.1.1 Técnicas de programação feita por usuários finais 8
- 2.1.2 Problemas e desafios 12

**2.2 Engenharia Semiótica** 16

**2.3 Engenharia Cognitiva e Aplicações Extensíveis** 24

**2.4 Representação do Conhecimento** 25
- 2.4.1 Representação dos Modelos do Domínio 26

**2.5 Analogias** 27
- 2.5.1 Metáforas e Metonímias 28
- 2.5.2 Analogias e Abdução 29
- 2.5.3 Metáforas e Semiótica 31
- 2.5.4 Analogias em Inteligência Artificial 31

**3. PROGRAMAÇÃO VIA INTERFACE** 35

**3.1 Visão Geral** 35
- 3.1.1 Extensão por Interpretação 37
- 3.1.2 Extensão por Construção via Wizards 37
- 3.1.3 Considerações 40

3.2 Representação do conhecimento 42

3.3 Geração de Metáforas e Metonímias 44
3.3.1 Eixos de Extensão 44
3.3.2 Classificações Utilizadas nos Cálculos de Metonímias 45
3.3.3 Classificações Utilizadas nos Cálculos de Metáforas 45

3.4 Mecanismos de extensão 46
3.4.1 Tipos de Extensão 46
3.4.2 Extensões em Tipos 49
3.4.3 Extensões em Operações 54

*3.5 Wizards* 57

3.6 Discussão 58
3.6.1 União de classificações existentes 61
3.6.2 Interseção de classificações existentes 61
3.6.3 Diferença entre classificações existentes 62
3.6.4 Especialização de uma classificação existente 62

4. PROGRAMAÇÃO VIA INTERFACE EM AÇÃO 63

4.1 O Protótipo 63

4.2 Modelo Estático 65

4.3 Modelo Dinâmico 66

4.4 Mapeamentos Expressão–Conteúdo 68

4.5 Situações de Uso 71

4.6 Orientações para o *Designer* 76

5. CONCLUSÕES 81

5.1 Discussão 81

5.2 Contribuições 84

5.3 Trabalhos Futuros 86

REFERÊNCIAS 88

APÊNDICE A — GLOSSÁRIO 98

APÊNDICE B — LISTAGENS 103

# Índice de Figuras

*Figura 1 — Processos estudados pela Engenharia Semiótica no caso de aplicações extensíveis.* 18

*Figura 2— Arquitetura de aplicação centrada na comunicação.* 19

*Figura 3 —Algumas telas de configuração de parâmetros em um editor de textos comercial.* 38

*Figura 4 — Tela de um editor de textos após ter sido customizada pelo usuário.* 40

*Figura 5 — Prefixos utilizados na linguagem de representação.* 42

*Figura 6 — Porção de tela ilustrando o mundo de Karel.* 64

*Figura 7 — Representação de parte do modelo estático do domínio do nosso protótipo.* 64

*Figura 8 — Seqüência de telas ilustrando parte da situação de uso da criação de um novo tipo a partir de um tipo existente* 73

*Figura 9 — Seqüência de telas ilustrando parte da seqüência de uso de criação de uma nova operação a partir de uma operação existente.* 76

# Índice de Tabelas


*Tabela 1 — Ilustração do uso dos prefixos que distinguem designer/usuário, e conteúdo/expressão, na linguagem de representação.* 43

*Tabela 2 — Exemplos de predicados sobre tipos definidos pelo designer da aplicação.* 43

*Tabela 3 — Exemplos de mapeamento conteúdo–expressão representados em nosso protótipo. Os mapeamentos estão agrupados de acordo com os valores de cada atributo.* 69

*Tabela 4 — Conjunto típico de* widgets *que podem ser utilizados no mapeamento conteúdo–expressão das operações e seus parâmetros.* 70

*Tabela 5 — Situação de uso do protótipo num exemplo onde o usuário deseja criar um novo tipo, a partir de um tipo existente.* 72

*Tabela 6 — Situação de uso do protótipo num exemplo onde o usuário deseja criar uma nova operação, a partir de uma operação existente.* 75

# Notação utilizada nesta tese

Os termos em língua estrangeira estão representados em *itálico*.

Um asterisco (*) ao lado de uma expressão indica um termo cuja definição pode ser encontrada no glossário, no Apêndice A.

Trechos de código e pseudo-código estão escritos em fonte sem serifa.

Formatação em **negrito** é utilizada para destacar algumas expressões ao longo do texto.

> Os exemplos apresentados neste trabalho estão representados em corpo menor, e com endentação diferenciada.

# Agradecimentos

Ao CNPq, pela ajuda financeira recebida durante o curso.

Aos colegas do Departamento de Informática, e em especial aos membros do *Semiotic Engineering Research Group*, pelos estudos em conjunto e discussões que tanto contribuíram para este trabalho.

À minha Orientadora, Professora Clarisse Sieckenius de Souza, pelo incentivo à boa formação acadêmica, não apenas pela exigência de um trabalho bem feito, mas principalmente por ser ela própria exemplo a ser seguido, e em especial pela dedicação em todos os momentos e nas mais diversas circunstâncias.

Aos amigos Karin, André, Viviana, Leonardo, Vera, dentre tantos outros, *for being there*.

A toda minha família, que sempre me apoiou e ajudou. Em especial aos meus pais, que ao longo de toda minha vida contribuíram para minhas realizações.

Aos meus filhos Gabriel e Eduardo, nascidos e criados em meio aos estudos e desenvolvimento desta tese, pela distração e alegria proporcionadas nos momentos de pausa, muitas vezes forçada, e também pela compreensão e bom comportamento nas muitas vezes em que me ouviram dizer: "a mamãe precisa trabalhar na tese", permitindo assim a conclusão deste trabalho. Agora podemos brincar.

Ao meu marido Luís Fernando, melhor amigo e companheiro de todas as horas, incansável, pelo incentivo a todo custo, pelas broncas necessárias ou merecidas, pela ajuda em todos os sentidos, por todos os sacrifícios, mas principalmente pelo amor, carinho e dedicação incomparáveis. Este trabalho é dedicado a você.

# 1. Introdução

A indústria de software vem ao longo dos anos aumentando a funcionalidade* das aplicações, numa tentativa de satisfazer as necessidades do maior número de usuários possível. Entretanto, essa solução não apenas não consegue atingir totalmente a meta, como ainda acarreta outros problemas tão graves quanto (ou mais do que) a falta de funcionalidade.

Os processos de *design* de software dedicam grande esforço na elicitação, análise e especificação de requisitos, de forma a maximizar a utilidade e eficiência do software resultante (Pressman, 1996). Entretanto, essa estratégia de solução ainda não garante a consecução total da meta, visto que os problemas dos usuários podem mudar com o tempo, e os engenheiros de requisitos e de software não têm como prever toda a evolução das tarefas dos usuários. Além disto, os usuários podem querer fazer um uso criativo da aplicação, utilizando-a para solucionar problemas para os quais a aplicação não havia sido projetada. Assim, vemos que mesmo as aplicações com muita funcionalidade não conseguem resolver todos os problemas de todos os usuários, o tempo todo.

Devido à natureza evolutiva das aplicações computacionais, os usuários estão sujeitos a utilizar diversas versões de uma mesma aplicação. Cada versão geralmente incorpora novas operações*, visando a atender uma classe maior de problemas. Para tornar essas novas operações disponíveis, devem ser criados também novos elementos de interface, desde itens de menu até botões e *toolbars*, quando não complexos padrões de manipulação direta. Com isso, o usuário é levado com freqüência a ter que reaprender a utilizar uma parcela da aplicação a cada nova versão. Com o passar do tempo, a interface pode se tornar tão complexa que o usuário não consegue mais absorver todo o modelo da aplicação*. Esta situação se torna ainda mais evidente em aplicações de propósito geral, como editores de texto e planilhas.

Qual seria então a solução? Muitas aplicações hoje em dia oferecem mecanismos de customização ou programação, transferindo para o usuário parte do problema de *design*, onde ele se torna também responsável pela eficiência da aplicação na resolução de seus problemas. Elas são o reflexo de uma tendência a querer aumentar, ao invés de substituir, as habilidades dos usuários (Adler e Winograd, 1992; Winograd, 1996). Essas aplicações suportam o que é chamado de *end-user programming*, ou programação feita por usuários finais. Um dos objetivos dessas aplicações é permitir que o próprio usuário programe a aplicação (Myers et al., 1992; Nardi, 1993; Cypher, 1993a; Eisenberg, 1995, Fischer, 1998). Técnicas de configuração de parâmetros, gravação de macros, de programação por demonstração e até mesmo linguagens de programação completas são oferecidas para os usuários finais estenderem suas aplicações. Chamaremos de **aplicação extensível** a aplicação que disponibilize um ou mais mecanismos que possibilitem sua extensão ou programação.

O *design* de aplicações extensíveis levanta algumas questões, entre outras: Quando o usuário vai querer programar? Ele pode aprender a programar? Caso possa, quais são suas dificuldades? A que se devem? As respostas para estas perguntas, como esta tese pretende mostrar, são no momento difíceis de dar. Porém, é possível fazer algumas observações facilmente comprováveis em nossa própria (e intensa) experiência cotidiana como usuários de software.

O usuário vai querer programar, ou estender, a aplicação quando perceber que precisa de uma operação que não está disponível. Para isto, no entanto, ele precisa conhecer a

aplicação o suficiente para saber que esta operação não pode ser realizada diretamente através da interação com a aplicação original*.

Embora possa parecer difícil que um usuário final qualquer seja capaz de aprender a programar enquanto esteja realizando sua tarefa, estudos indicam que os pré-requisitos para se escrever ou descrever procedimentos semelhantes a programas (e.g. dominar um código formal, ter capacidade de formular um plano de ações, imaginar diferentes cenários em que modificações são feitas nos planos) não são extraordinários para a média dos usuários. Em particular, Nardi aponta diversas linguagens formais com as quais as pessoas lidam no dia a dia, como aritmética, tabelas de jogos e receitas de tricô, entre outras (Nardi, 1993). Ela conclui que a dificuldade dos usuários em programar suas aplicações está nas linguagens de programação que lhes são oferecidas. Assim, parte da pesquisa em programação feita por usuários finais deve destinar-se a projetar linguagens de programação mais adequadas a serem oferecidas pelas aplicações extensíveis, para os usuários finais que desejem estendê-las.

Nossos estudos (de Souza e Barbosa, 1996; Barbosa et al., 1997; da Silva et al., 1997, Barbosa et al., 1998) indicam que uma linguagem de programação adequada pode não ser o bastante para resolver o problema. A interface das aplicações extensíveis também deve ser cuidadosamente projetada para facilitar não apenas a interação com a aplicação, mas agora também sua extensão.

Nossa abordagem parte do princípio que os usuários podem querer estender a aplicação partindo de um determinado objeto ou operação, mudando apenas algumas de suas propriedades ou parte de seu comportamento. Nesta hipótese, o usuário pensaria da seguinte forma: "*Se houvesse uma operação 'como essa' mas que pudesse ser aplicada sobre 'aquele objeto'...*" ou "*Se houvesse um objeto 'como esse' mas que possuísse mais 'tais' propriedades...*". Fica evidente que esta maneira de raciocinar se baseia na percepção de semelhanças e na elaboração de analogias* que, lingüisticamente, se expressam preferencialmente através de metáforas* e metonímias*. De forma a ajudar os usuários finais a realizar extensões simples, mas que não são possíveis apenas através de mecanismos de configuração e de gravação de macros, nossa abordagem utiliza justamente estudos sobre analogia para calcular possíveis extensões e sugerir opções de extensão com base em alguma indicação prévia do usuário sobre o foco da extensão, ou seja, operação ou objeto que ele deseja estender.

Nosso trabalho faz parte de um conjunto abrangente de trabalhos dentro da área de Engenharia Semiótica (de Souza, 1993), cujo objetivo maior é oferecer meios de melhorar a comunicação entre *designer* e usuários. Consideramos principalmente as linguagens com as quais os usuários devem interagir. De acordo com a Engenharia Semiótica, vemos que linguagens adequadas promovem melhor usabilidade da aplicação resultante.

No caso específico de aplicações extensíveis, o usuário deve lidar não apenas com uma linguagem de interface, mas também com algum tipo de linguagem de programação ou extensão, o que envolve considerações adicionais e requer expansões do quadro teórico de Engenharia Semiótica. Essencialmente, o ponto central da expansão está em o usuário passar a exercer o papel de (algum tipo de) *designer* e de, conseqüentemente, o *designer* de aplicações extensíveis ter de realizar uma espécie de *meta-design*, ou *design* para o *design*.

Em outras palavras, enquanto as interfaces das aplicações que não podem ser estendidas devem considerar a curva de aprendizado do usuário para facilitar a transição de usuário novato para usuário experiente, em nosso caso temos também que facilitar a transição de usuário experiente–*designer* novato para usuário experiente–*designer* experiente. Vale observar aqui que não pretendemos transformar os usuários em programadores, mas oferecer-lhes um ferramental que lhes possibilite assumir o papel de solucionadores de problemas, seja envolvendo programação ou não. Por esta razão, preferimos nos referir a "extensão" ao invés de "programação", ao citarmos os mecanismos e linguagens envolvidos nesse processo.

As linguagens de extensão devem ser cuidadosamente projetadas para permitir seu uso por usuários leigos em programação. Uma das pesquisas de nosso grupo consiste em projetar uma linguagem de programação mais "natural" aos usuários finais (da Silva e Ierusalimschy, 1997; Barbosa et al., 1999). Entretanto, ainda assim podemos considerar longo o caminho que os usuários devem percorrer para realizar suas tarefas, desde o aprendizado sobre como interagir com a aplicação, passando por como interagir com um sub-ambiente de extensão, aprender conceitos básicos de programação, descobrir o modelo subjacente da aplicação ou da parte que se quer estender e, finalmente, aprender como utilizar todo este conhecimento para resolver o problema atual.

Nosso trabalho apresenta um modelo de aplicação extensível que traz para a linguagem de interface mecanismos de extensão que fazem uso de alguns recursos semântico–pragmáticos, a saber: metáforas* e metonímias*. Através da interface, apresentamos os modelos do domínio e da aplicação, permitimos que os usuários indiquem o ponto de partida das extensões desejadas, e os guiamos passo a passo através destas extensões, utilizando mecanismos do tipo *wizard**. Como as extensões são geradas por cálculos metafóricos e metonímicos, e sugeridas aos usuários, que têm a palavra final sobre a extensão resultante, o *designer* consegue manter consistência com o significado pretendido da aplicação. Isto é obtido através de restrições nas extensões, tanto da funcionalidade quanto da interface da aplicação.

Ao trazer estes recursos de extensão para a interface, evitamos problemas de descontinuidade entre interação e extensão, e permitimos que usuários leigos em programação façam extensões nas aplicações sem precisar primeiro aprender conceitos ou linguagens de programação. Para tanto, o modelo prevê que a aplicação revele gradualmente os modelos de domínio e da aplicação, à medida que forem necessários para as extensões (DiGiano e Eisenberg, 1995; DiGiano, 1996).

Em outras palavras, nossa abordagem tem como resultado dar aos usuários recursos para estender os modelos criados pelo *designer* da aplicação, de forma controlada. Ela quebra barreiras entre interface e ambiente de extensão, incorporando à linguagem de interface mecanismos meta-lingüísticos de extensão, de natureza semântico–pragmática, a saber: metáforas* e metonímias*.

Embora nosso modelo considere a linguagem de interface, e não a linguagem de extensão propriamente dita, ele prevê que, caso haja uma linguagem de extensão disponível aos usuários, esta lhes seja gradualmente revelada, juntamente com os modelos do domínio e da aplicação. Assim, promovemos uma capacitação gradual do usuário, que vai desde a interação, passando pelas extensões que utilizam os recursos metalingüísticos, até, possivelmente, a geração manual de código, escrito na linguagem de extensão, e em um ambiente apropriado, que chamamos de sub-ambiente de extensão de uma aplicação.

Nosso trabalho pode ser utilizado para complementar abordagens que modelam o conhecimento e aprendizado do usuário, como em [Schmalhofer et al., 1995; Weber e Bögelsack, 1995; Witschital, 1995]. Enquanto estas abordagens se baseiam em modelos

cognitivos e tentam tratar do processo de compreensão e aprendizado humano, a nossa se concentra nos aspectos semióticos ou de comunicação entre interface e usuários, mais especificamente no potencial comunicativo de metáforas e metonímias para extensão de software.

O ponto central desta tese é a introdução de um novo paradigma de programação feita por usuários finais, que chamamos de **programação via interface**, inspirado em duas fontes básicas: no cálculo de metáforas e metonímias, e nas noções de Abstração Interpretativa e Contínuo Semiótico da Engenharia Semiótica, como será visto ao longo desta tese.

O Capítulo 2, "Fundamentos Teóricos", apresenta fundamentos teóricos relacionados ao nosso trabalho, das áreas de Computação, Semiótica e Ciência Cognitiva. Nosso modelo é descrito no Capítulo 3, "Programação Via Interface". O Capítulo 4, "Programação via Interface em Ação", apresenta um contexto de uso do nosso modelo, exemplificado através de um protótipo. Finalmente, no Capítulo 5, "Conclusões", comparamos nossa abordagem com trabalhos relacionados, resumimos nossas contribuições e apontamos direções para trabalhos futuros. Como material de suporte, esta tese inclui dois apêndices: um glossário, no Apêndice A, que descreve os termos utilizados ao longo da tese, e o Apêndice B, que contém listagens em Prolog que ilustram os mecanismos apresentados ao longo desta tese.

# 2. Fundamentos Teóricos

O objetivo deste capítulo é apresentar os conceitos utilizados neste trabalho e em trabalhos relacionados. Em particular, descrevemos algumas técnicas de programação feitas por usuários finais, apresentando definições, problemas e desafios encontrados em aplicações extensíveis. Em seguida, apresentamos os conceitos de Engenharia Semiótica e como são utilizados para auxiliar o *designer* de uma aplicação a atingir seu objetivo comunicativo, ou seja, facilitar o entendimento e uso da aplicação pelo usuário. Finalmente, apresentamos estudos sobre analogias e sua aplicação na área de Ciência da Computação, em particular em Inteligência Artificial. Estes estudos têm a função de mostrar o potencial de uso dos mecanismos analógicos para extensão de software, e a viabilidade da implementação de alguns destes mecanismos no ambiente computacional.

## 2.1 Aplicações extensíveis

Chamamos **aplicações extensíveis** as aplicações que podem ser de alguma forma configuradas ou estendidas pelos usuários finais. Em uma aplicação extensível, chamamos de sub-ambiente de interação as partes da aplicação com as quais o usuário

interage sem intenção de estendê-la, e de sub-ambiente de extensão as partes da aplicação com as quais o usuário interage durante o processo de extensão da mesma. Uma aplicação extensível pode funcionar no **modo de interação** ou no **modo de extensão**, estados em que o usuário interage com o sub-ambiente de interação ou com o sub-ambiente de extensão, respectivamente. Como resultado de uma extensão, teremos ainda o sub-ambiente de interação estendido, ou seja, o ambiente com o qual o usuário passa a interagir após ter feito uma extensão.

### 2.1.1 Técnicas de programação feita por usuários finais

Esta seção descreve técnicas que permitem que usuários criem seus próprios comandos personalizados. Essas técnicas são reunidas sob o selo de "programação feita por usuários finais", embora nem todas envolvam uma atividade de programação propriamente dita. Utilizaremos a classificação das técnicas de programação para usuários finais apresentada em [Cypher, 1993a].

#### Configuração de parâmetros

Nesta técnica, são apresentadas diversas alternativas predefinidas que o usuário pode selecionar de acordo com sua preferência. Uma aplicação extensível que utiliza esta técnica pode ser analisada de acordo com o número de parâmetros que o usuário pode configurar e o número de combinações obtidas. Poucos parâmetros e combinações resultam em um baixo potencial de extensão da aplicação, enquanto muitos parâmetros reduzem as chances de o usuário prever com sucesso qual será a configuração resultante da combinação desses parâmetros, limitando seu uso.

Embora possa ser um mecanismo de programação poderoso do ponto de vista computacional — vejam-se as redes neurais, por exemplo —, seu emprego por usuários finais é naturalmente muito limitado, visto que o número de parâmetros que se pode esperar que os usuários entendam e configurem é bastante pequeno.

O grande desafio para utilização dessa técnica é descobrir quais e quantos parâmetros poderão ser configurados pelos usuários, a fim de garantir duas coisas. Uma é a satisfação de suas necessidades individuais sem que haja um número excessivo de parâmetros e combinações diferentes. A outra é um efeito consistente das extensões sobre o significado

e a retórica global da aplicação tal como inicialmente projetada pelo *designer* (ou seja, impedir que uma extensão desdiga o que o *designer* disse em primeiro lugar).

### Gravação de macros

Esta técnica permite que os usuários ativem um gravador de instruções, que registra cada passo de interação dentro de uma seqüência de comandos. Esta seqüência é armazenada e pode ser reproduzida quando necessário. Embora seja uma técnica bastante popular, ela é “literal demais”, ou seja, grava literalmente os eventos de interface situados no contexto de gravação, tendo como constantes os valores das variáveis de contexto.

Uma macro resultante de gravação é uma seqüência linear de instruções. Ela representa um autômato finito, cuja classe de problemas resolvíveis é bastante restrita (Chomsky, 1959; Hopcroft e Ullman, 1979). Para contornar estas limitações, muitas das aplicações que permitem gravar instruções em uma macro oferecem também mecanismos de edição de macros.

> Aplicação: Microsoft® Word© for Windows 2.0. O usuário possui um PC em casa e um Macintosh no trabalho, e precisa freqüentemente salvar o documento para Mac para levar para o trabalho. Abre o arquivo desejado, CARTA1.DOC, e grava uma macro para armazená-lo no disquete e no formato que possa ser lido no editor de textos do Mac, com extensão MCW. Em seguida, abre o arquivo PROJETO.DOC e pede para executar a macro, imaginando que será criado um arquivo PROJETO.MCW no disquete. Entretanto, o editor pergunta ao usuário se deseja sobrescrever o arquivo CARTA1.MCW existente no disquete. O usuário responde ‘não’, e desiste da macro.

O caso descrito acima ilustra uma das limitações da técnica de gravação de macros: o que o usuário digita ou seleciona com o mouse é gravado literalmente. No diálogo de interação, quando o usuário seleciona o tipo de arquivo, a aplicação toma como ponto de partida o nome do documento atual, e acrescenta a extensão escolhida. Ao gravar esta interação, entretanto, apenas o nome resultante é gravado, como uma constante. Algumas aplicações contornam esse problema permitindo que, ao reproduzir as operações gravadas, sejam apresentados aos usuários os quadros de diálogo para que eles possam alterar esses valores.

Nas aplicações que permitem editar macros, as alterações podem ser feitas através de diálogos ou de edição direta do código gravado. A edição por diálogos permite contornar

a limitação do uso de constantes, visto que alguns valores podem ser preenchidos pelo usuário em tempo de execução, mas não supera a limitação da linearidade do código. Em contrapartida, a edição direta do código pode oferecer aos usuários todos os recursos de linguagens de *script* ou de programação, o que acarreta outros problemas, como será visto a seguir.

### Linguagens de programação

Devido às limitações de gravação de macros, muitas aplicações extensíveis disponibilizam verdadeiras linguagens de programação para os usuários finais. Algumas aplicações apresentam linguagens com menos recursos, e cujos itens lexicais são tomados da linguagem natural, visando facilitar a compreensão. Estas são chamadas linguagens de *script*.

Nas aplicações que oferecem linguagens de *script* ou de programação para os usuários finais, geralmente encontramos uma mudança no modo de operação, ou até um ambiente diferente para a extensão da aplicação. O que ocorre comumente é uma mudança drástica na interface da aplicação quando o usuário escolhe fazer uma extensão, onde lhe é oferecido um editor de programa, textual, na maioria das vezes. Assim, o usuário final encontra-se desorientado, não sabendo por onde começar, como interagir com esta "nova aplicação".Além disso, os usuários finais geralmente são leigos em programação, e precisam de suporte não apenas para compreensão dos modelos do domínio[1] e da aplicação, e da interação com a mesma, mas também para o aprendizado de conceitos e técnicas de programação em geral.

Outro problema é a própria linguagem de programação. De acordo com Nardi, uma linguagem de programação para usuários finais deve ser específica para a tarefa do usuário (Nardi, 1993). De acordo com a autora, linguagens de *script* como definidas em [Cypher, 1993a] seriam então mais adequadas aos usuários finais.

Outro tipo de linguagem de programação que consideramos foram as linguagens de programação visuais (Chang, 1990), utilizadas principalmente para lidar com operações espaciais. Estas linguagens privilegiam aplicações cujo domínio apresente um

[1] Embora os usuários geralmente conheçam o domínio da aplicação, não sabem como este foi modelado pelo *designer*. Desta forma, uma aplicação extensível precisa também dar suporte à revelação deste modelo para o usuário.

mapeamento dos elementos perceptivos da linguagem visual e seus atributos estáticos (cor, forma, tamanho) para os elementos do domínio. Já em relação às operações, este tipo de linguagem privilegia as operações de natureza espacial, como ordenação, translação, rotação, e assim por diante. Isto explica porque grande parte dos trabalhos em [Chang, 1990] são aqueles que utilizam como representação visual grafos, árvores e manipulações sobre estas estruturas.

Elementos que possuem uma realização concreta são mais adequados a representações visuais. Entretanto, conceitos abstratos que não possuem uma representação gráfica convencionada tornam-se um exercício de interpretação, que pode dificultar ou impossibilitar a tarefa do usuário. Este problema se deve principalmente à baixa expressividade de linguagens visuais, quando comparadas às textuais (Martins, 1998). Em aplicações extensíveis, torna-se fundamental lidar com os tipos abstratos, e não suas instâncias, o que torna ainda mais difícil representá-los visualmente. Além disto, os usuários finais encontram os mesmos problemas de linguagens de programação tradicionais, ou seja, a necessidade de conhecer conceitos e técnicas de programação e computação em geral.

Em todos os estilos de linguagem de programação, os usuários finais precisam lidar com linguagens computáveis. No caso de linguagens imperativas, eles precisam aprender a manipular variáveis, condicionais e iterações. No caso de linguagens funcionais, o usuário deve estar familiarizado com aplicação de funções. No caso de linguagens lógicas, praticamente precisa conhecer lógica de primeira ordem.

### Programação por demonstração

Programação por demonstração se assemelha à técnica de programação por gravação de macros, mas ao invés de a aplicação registrar literalmente as operações ativadas pelo usuário, ela cria programas generalizados a partir destas operações, e os programas resultantes podem conter iterações e condicionais. A idéia de se definir problemas de forma operacional através de exemplos foi proposta por Zloof através do sistema de recuperação de informações de bancos de dados QBE, *Query By Example*, cujo usuário-alvo era aquele com pouca ou nenhuma experiência em programação ou em matemática (Zloof, 1981). Hoje em dia, existem diversos tipos de aplicações que se encaixam na definição de programação por demonstração, e elas podem ser subdivididas em aplicações

com ou sem mecanismos de inferência, ou diferir no tipo de construtos de programa a que dão suporte, ou no tipo de conhecimento embutido na aplicação (Cypher, 1993a).

> Em um editor de textos, existe um mecanismo que tenta adivinhar quando o usuário deseja criar uma lista numerada. Basta digitar um número seguido de 'tab' e algum texto adicional que, ao pressionar a tecla 'Enter', o editor de texto automaticamente insere o número consecutivo ao digitado. Supondo que o usuário deva digitar diversos dados cuja primeira informação seja um número, sua tarefa é interrompida a cada novo parágrafo, devido a uma opção habilitada por *default*, e cuja desativação não é de fácil acesso.

Um dos problemas da técnica de programação por demonstração é a necessidade de o usuário ter que exemplificar sua intenção através de diversos exemplos, numa tentativa de enumerar todas as situações que possam ocorrer, para que os mecanismos de inferência produzam um bom resultado, ou seja, uma extensão que se assemelha àquela pretendida pelo usuário. De modo geral, não lhe são apresentados claramente os modelos subjacentes do domínio ou da aplicação. Em outras palavras, uma aplicação projetada para programação por demonstração privilegia a interação com o usuário no sentido usuário → aplicação.

Como será visto ao longo deste trabalho, as aplicações projetadas para programação via interface consideram os aspectos comunicativos do usuário com a aplicação nos dois sentidos: o usuário expressa suas intenções através da interface, e a aplicação, por sua vez, revela os modelos embutidos pelo *designer* e as regras que governam os mecanismos de inferência da aplicação. Através deste ciclo de comunicação, o usuário se torna mais capacitado em prever as inferências que a aplicação pode fazer, e assim pode se expressar de maneira a maximizar a eficiência dos mecanismos de inferência, obtendo melhores extensões.

### 2.1.2 Problemas e desafios

A análise de algumas aplicações extensíveis de propósito geral, como editores de texto, planilhas e editores gráficos, revela que todas oferecem mecanismos de gravação de macro, e algumas permitem que o usuário escreva código diretamente em uma linguagem de programação textual.

Ao fazermos tal análise, tentamos identificar problemas específicos a aplicações extensíveis, desconsiderando problemas encontrados em aplicações que não podem ser estendidas. Procuramos:

- descontinuidades e inconsistências entre as interfaces dos sub-ambientes de interação e de extensão;
- descontinuidades e inconsistências entre a linguagem de interface e a linguagem de programação;
- descontinuidades e inconsistências entre o sub-ambiente de interação antes da extensão e o mesmo após a extensão.

Na análise de editores de texto, foram escolhidas as funções de busca e substituição, de armazenamento de arquivos e de *undo*. Estas funções foram escolhidas por serem representativas de problemas de manipulação de *strings*, de problemas de nome de espaços de usuário, e de controle de execução. Comparou-se o acionamento destas funções através de interação na interface com o código resultante da gravação em macro, bem como os parâmetros que os usuários podem ou devem configurar em cada caso.

Embora a análise tenha sido feita por pós-graduandos da área de Computação (Barbosa et al., 1997), diversos problemas e dificuldades foram detectados quando eles próprios tentaram construir extensões, o que sugere que seria ainda mais difícil para usuários leigos em programação realizarem suas tarefas utilizando os mecanismos de extensão oferecidos por estas aplicações.

Um dos problemas encontrados nas aplicações extensíveis analisadas foi a descontinuidade entre os sub-ambientes e linguagens de interação e de extensão.

Ao ativar uma operação de extensão de gravação de macro, muitas vezes ocorre apenas uma mudança no estado ou modo da aplicação, com pouca mudança perceptiva, como uma simples alteração no cursor do mouse, o que leva o usuário a crer que pode continuar a interagir com a aplicação como de costume. Entretanto, algumas operações podem não estar disponíveis ao usuário no modo de extensão, sem que isto lhe seja indicado.

> Aplicação: Microsoft® Word© for Windows 2.0. O usuário deseja copiar um pedaço de texto e colá-lo em diversos locais do texto. Ao ativar a operação de gravação de macro, o cursor do mouse é alterado para indicar a mudança de modo, e aparece uma *toolbar* com botões para interromper ou encerrar a gravação. O usuário tenta selecionar o pedaço de

texto que quer copiar utilizando o mouse, mas não consegue. Nada acontece, nenhuma mensagem lhe é apresentada. O usuário acessa o menu com o mouse para ativar a opção de ajuda, e após alguma insistência, encontra a informação de que, no modo de gravação, a seleção do texto com o mouse não funciona, e que é necessário utilizar o teclado.

Este exemplo ilustra a falta de uma sinalização clara da mudança do modo de interação para o modo de extensão, no que diz respeito à maneira como o usuário pode acessar determinadas operações. A operação de seleção de texto está disponível tanto no modo de interação quanto no de extensão, mas uma das maneiras de acessá-la não está disponível no modo de extensão. Para um usuário que não está acostumado a utilizar o teclado como ferramenta de seleção, a tarefa de selecionar um texto muito grande pode se tornar bastante trabalhosa.

Após uma análise mais cuidadosa, percebemos que lidamos de fato com duas operações distintas: uma operação de seleção relativa à posição atual do cursor, utilizando as setas do teclado, e uma operação de seleção utilizando o mouse. Ao ativar a operação de gravação, a operação de seleção através do teclado é mantida, mas a de seleção com o mouse não está mais disponível.

Este caso nos mostra que, embora exista uma regra ou explicação para uma mudança entre os modos, o fato de não deixar isto claro através da interface gera dúvidas para o usuário, que pode perceber a mudança como um comportamento inconsistente da aplicação.

Outro problema encontrado com freqüência em aplicações extensíveis é a mudança no comportamento acionado por um signo* de interface durante a interação ou durante uma extensão.

Aplicação: Microsoft® Word© for Windows 2.0. O usuário está acostumado a interagir com a aplicação e, quando erra, ativa a opção de *undo* e desfaz o último passo de interação. Após gravar uma macro, o usuário decide testá-la. Vendo que o resultado não foi o desejado, decide ativar o *undo*. Entretanto, o documento não foi restaurado ao seu estado anterior à execução da macro. Apenas o último comando da macro foi desfeito. Em alguns casos, este comportamento pode não ser percebido, o usuário pode pensar que todas as operações da macro foram desfeitas, e o resultado pode ser desastroso.

Esse caso ilustra como o comportamento de um elemento de interação pode ter seu comportamento alterado, pois um único passo de interação pode ser mapeado para mais de uma operação, e assim teremos comportamento inconsistente do elemento de interação que ativa esta operação.

Quando há possibilidade de escrever diretamente código em uma linguagem de extensão ou editar um código previamente gravado, um problema freqüente é a descontinuidade entre a linguagem de interface e a linguagem de extensão, como pode ser visto no caso abaixo.

> Aplicação: Microsoft® Word 97©. Um usuário experiente grava uma macro para fazer uma busca e substituição no documento. Ao gravá-la, interage com um diálogo que tem como parâmetros, entre outros, a direção da busca (*Search: All, Down* ou *Up*). Seleciona *Down*, e realiza a substituição. Somente após gravar a macro, verifica que deveria ter selecionado *All*. Resolve então editar a macro recém-criada. Procura o parâmetro *Search*, mas não encontra. Em vez disso, nota o parâmetro *Forward* que não existia no diálogo com o qual interagiu. Devido à semelhança com a língua inglesa, ele imagina que *Forward* indique a direção da busca, mas seu valor não é o valor *Down* selecionado através do diálogo, mas sim *True*. Inspecionando mais ainda, pode encontrar o parâmetro *Wrap*, e só então tentar identificar que o parâmetro *Search = Down* do diálogo é mapeado para a combinação de parâmetros *Forward = True* e *Wrap = wdFindContinue*.

Neste caso, houve uma quebra entre a linguagem de interface, ou seja, os elementos do diálogo com os quais o usuário estava acostumado a interagir, e a linguagem de extensão, ou seja, os parâmetros da operação que ele deve manualmente digitar para realizar a operação. Este tipo de descontinuidade dificulta e, em certos casos, impossibilita a edição pelo usuário final do código resultante da gravação de uma macro.

Algumas linguagens de extensão tentam solucionar os problemas apresentados importando abstrações de outras linguagens de programação, o que as torna ainda mais difíceis de utilizar. Como descrito por Ghezzi e Jazayeri: "*...the proliferation of built-in abstractions, as opposed to mechanisms for defining new abstractions, has the principal effect of making the language large without exhausting all the needs that may arise in different applications*" (Ghezzi e Jazayeri, 1987).

A atividade de programação envolve, a grosso modo, uma linguagem de programação e um ambiente de desenvolvimento. Este ambiente deve oferecer documentação tanto sobre

a linguagem de programação quanto sobre sua interface. Esta documentação deve ir além de manuais on-line, e incluir explicações sobre o funcionamento da aplicação, revelando a intenção de *design* do *designer* original. Como levantado por [da Silva et al., 1997], este mecanismo de explicação, integrado aos ambientes de interação e de extensão, não é oferecido.

A extensão de uma aplicação geralmente pressupõe um ambiente de desenvolvimento embutido na própria aplicação. Entretanto, as interfaces da aplicação e deste ambiente de desenvolvimento são geralmente independentes e muito diferentes. Os usuários devem atravessar um golfo enorme ao mudar de um contexto para outro: de "utilizar" para "estender" a aplicação.

Além disso, o sub-ambiente de extensão freqüentemente desconsidera a falta de experiência do usuário final em programar e a falta de conhecimento de conceitos de computação em geral. O que encontramos com freqüência não passa de um editor de texto e uma lista de comandos e suas estruturas sintáticas, para que o usuário escreva o código desde o princípio, além de algumas ferramentas de acompanhamento da execução e depuração do código escrito. O resultado é que os usuários não sabem nem por onde começar.

Nosso trabalho oferece um modelo que traz para a linguagem de interface, e ao alcance de usuários leigos em programação, mecanismos de extensão que fazem uso de operadores meta-lingüísticos para possibilitar alguns tipos de extensão sobre a aplicação. Utilizamos a Engenharia Semiótica como principal fundamento teórico de nossa abordagem, como será descrito a seguir.

## 2.2 Engenharia Semiótica

Esta seção apresenta conceitos da Engenharia Semiótica, que utilizamos como base teórica para nosso trabalho, e a arquitetura de aplicação proposta em [de Souza, 1997]. O objetivo é promover uma melhor compreensão deste trabalho, descrevendo o contexto no qual ele se encaixa, como a abordagem semiótica contribui para o desenvolvimento de aplicações extensíveis, e enumerar questões teóricas de fundo que são discutidas ao longo desta tese.

A Engenharia Semiótica investiga aplicações de software utilizando uma perspectiva de comunicação (de Souza, 1993). Em primeiro lugar, há uma comunicação unilateral, do *designer* para o usuário, onde o *designer* tenta comunicar ao usuário sua interpretação sobre o problema deste, e que solução ele, o *designer*, lhe oferece. Além disto, a própria aplicação é vista como um agente comunicativo, que troca mensagens com o usuário. Entretanto, a aplicação é um agente muito especial, que emite exatamente as mesmas mensagens toda vez que se encontra na mesma situação.

Um dos objetivos da Engenharia Semiótica é tornar os usuários conscientes de que as aplicações são produtos de um ser humano que nelas fixou sua interpretação para determinados problemas e tarefas dos usuários, e algumas soluções para estes problemas.

### Abordagem centrada na comunicação

Aplicações computacionais podem ser vistas como artefatos de meta-comunicação, pois são mensagens do *designer* aos usuários, mensagens estas que enviam e recebem outras mensagens do usuário. Esta é a idéia central da Engenharia Semiótica (de Souza, 1993; de Souza, 1996; de Souza, 1997). Trata-se de uma abordagem com base teórica para o *design* de códigos e linguagens de interface de usuário. A Engenharia Semiótica defende que grandes avanços ocorrerão quando os *designers* perceberem que eles (e não a aplicação que eles projetam) estão se comunicando com os usuários através da interface da aplicação, e os usuários, por sua vez, perceberem que não estão recebendo uma mensagem de um sistema ou de uma máquina, mas sim do *designer*.

Para que os usuários sintam a necessidade e possam estender uma aplicação, eles precisam inicialmente entender seu significado, e como novos significados podem ser acrescentados a ela. A Engenharia Semiótica pretende facilitar estes processos de interpretação do usuário através do *design* cuidadoso das linguagens envolvidas na aplicação. Além de decifrar as intenções do *designer*, os usuários devem agora poder expressar suas próprias intenções. Pretendemos motivá-los a utilizar este recurso comunicativo para escrever suas próprias mensagens. A Figura 1 ilustra os processos que ocorrem em aplicações extensíveis.

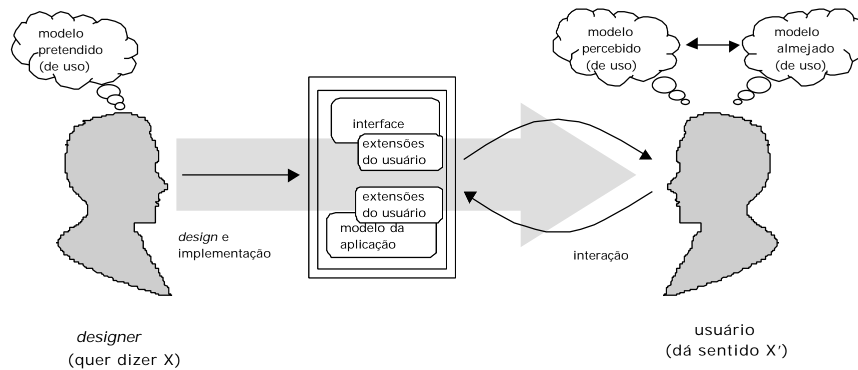


*Figura 1 — Processos estudados pela Engenharia Semiótica no caso de aplicações extensíveis.*

O modelo **pretendido** de uso indica como o *designer* percebe que o usuário vai utilizar a aplicação, ou seja, é a interpretação do *designer* para a solução dos problemas do usuário, ou melhor, da sua interpretação destes problemas. Já o modelo **percebido** de uso indica como o usuário percebe que pode utilizar a aplicação, embora não seja necessariamente o modelo **almejado** de uso, ou seja, como o usuário gostaria de utilizá-la.

De acordo com de Souza, as regras e modelos arbitrários subjacentes às aplicações são raramente comunicados aos usuários (de Souza, 1997). Nossa abordagem centrada na comunicação expõe a existência de códigos que podem ser aprendidos, e permite revelar as regras gramaticais e semânticas em níveis sucessivos, para que os usuários possam fazer uso destes recursos lingüísticos para construir um novo discurso, e assim estender as aplicações.

Uma aplicação extensível deve fornecer então um ambiente de aprendizado para os usuários. Alguns componentes importantes em tal ambiente são interfaces multimodais, sistemas baseados em conhecimento e de explicação, módulos de ajuda e documentação, e ambientes de programação textuais e gráficos. Uma arquitetura assim é ilustrada pela Figura 2 (de Souza, 1997).

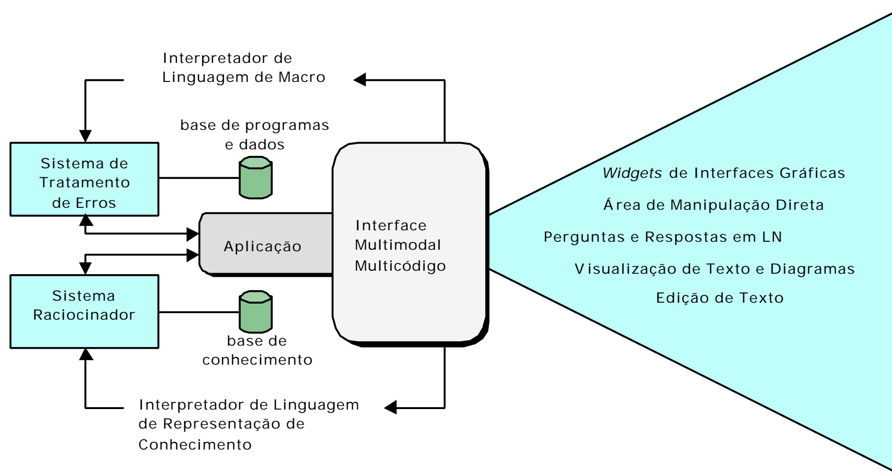


*Figura 2— Arquitetura de aplicação centrada na comunicação.*

Com exceção de algumas aplicações de programação por demonstração, as aplicações extensíveis com freqüência não possuem um subsistema baseado em conhecimento, que é parte integrante da nossa arquitetura, como pode ser visto no canto inferior esquerdo da figura. No canto superior esquerdo, temos um subsistema de interpretação de programas e tratamento de erros que fazem uso de uma base de programas e dados, que por sua vez é o repositório das extensões do usuário, acrescidas à base de forma modular.

Nesta arquitetura, os processos de comunicação se dão através de atividades integradas de programação e explicação. O objetivo é estender o conceito de co-referencialidade lingüística do âmbito de interfaces para o âmbito das aplicações como um todo. Quando duas linguagens são co-referenciais, uma expressão em uma destas linguagens possui uma expressão correspondente na outra, e vice-versa. Quando os usuários entram em um processo de extensão da funcionalidade da aplicação, os mecanismos de extensão devem requisitar explicações de forma esquemática, a fim de preservar os objetivos comunicativos aqui delineados. Em outras palavras, cada extensão deve possuir uma expressão correspondente na linguagem de explicação.

Em suma, a programação centrada na comunicação pode ser entendida como um esforço do *designer* para contar aos usuários o que ele tinha em mente ao projetar a aplicação, e assim aumentar as chances de ser bem compreendido, aumentando a usabilidade, utilidade e extensibilidade da aplicação.

Nosso trabalho se concentra no ciclo inferior ilustrado na Figura 2, que realiza raciocínios sobre as indicações feitas através da interface e sobre uma base de conhecimento, de forma a estender a aplicação. Vale notar que esta arquitetura é ideal, servindo como referência para o protótipo que desenvolvemos.

### Bases de conhecimento

Utilizamos um modelo estático e um modelo dinâmico para representar o conhecimento necessário aos mecanismos de cálculo de extensão. A distinção entre modelo estático e modelo dinâmico pode ser encontrada sob diversas formas na literatura acadêmica. Em Linguagens de Programação, Gelernter e Jagannathan apresentam *space-maps* e *time-maps* (Gelernter e Jagannathan, 1990). Em Engenharia de Software e IHC, temos modelo de domínio e modelo de tarefas (Pressman, 1996; Preece et al., 1994). Em Lingüística: gramática e pragmática (Lyons, 1981). Todos estes tentam tratar da questão das distinções que emergem entre as coisas em abstrato e as coisas em situação. Além disto, fazemos também a distinção entre expressão e conteúdo, inspirada em [Hjelmslev, 1963]. Aplicado ao ambiente computacional, um *token* expressivo é a realização, na interface, de um *token* de conteúdo.

Assim, as bases de conhecimento previstas na arquitetura que supomos representam:

- **modelo estático do domínio**: a natureza classificatória dos elementos do domínio e suas relações estáticas. Aqui estão representados não apenas os tipos de objetos do domínio, mas também uma taxonomia dos atributos destes tipos. Por exemplo, em nosso exemplo de aplicação para esta tese (Seção 4.1, "O Protótipo"), color e shape são atributos do tipo appearance, ou seja, classification(appearance, [color,shape]). Além dos atributos, representamos neste modelo a estrutura ou composição dos tipos, através de relações do tipo part-of.
- **modelo dinâmico do domínio**: interações entre os elementos do domínio, no tempo e para um determinado fim. São representados como uma seqüência de instruções, e através de pré- e pós-condições. Estes elementos (instruções, pré e pós-condições) também podem ser classificados, de forma a permitir a aplicação dos operadores de extensão, como será visto mais adiante.
- **modelo da aplicação**: interações entre alguns elementos do domínio e do sistema, necessários para a implementação da funcionalidade da aplicação e de sua interface.

Neste modelo ficam representados os mapeamentos entre os objetos do domínio (**conteúdo**) e sua realização na interface (**expressão**), definidos extensional e/ou intensionalmente[2].

A arquitetura descrita anteriormente pressupõe a existência de uma linguagem de interface com o usuário, uma linguagem de programação feita por usuários finais e uma linguagem de representação de conhecimento. A próxima subseção apresenta características desejáveis destas linguagens.

### Linguagens envolvidas em aplicações extensíveis

Como estamos lidando com diversos tipos de representação (interface, programas e explicações), não podemos deixar de investigar também os mecanismos semióticos que permeiam essas representações. Estudamos mecanismos comunicativos com base na teoria Semiótica (Andersen, 1990; Andersen et al., 1993; Nadin, 1988), e mais especificamente, com base na Engenharia Semiótica (de Souza, 1993; de Souza, 1996), que nos permitem projetar melhores linguagens de interface, de programação e de explicação. Com isto, pretendemos tratar parte da crise descrita por Nöth, de que tanto filósofos quanto cientistas da área de Computação utilizam e discutem "representações" sem qualquer fundamentação semiótica para o conceito, o que enfraquece sua argumentação (Nöth, 1997).

Nossa pesquisa considera a presença de três linguagens diferentes em aplicações extensíveis:

- **linguagem de interface** (UIL — User Interface Language),
- linguagem de programação para usuários finais ou **linguagem de extensão** (EUPL — End-User Programming Language), e
- linguagem de documentação e/ou **linguagem de explicação** (ExpEU — Explanations for End Users).

Ela visa dar subsídios para o *design* cuidadoso destas linguagens, a fim de facilitar a compreensão dos usuários finais sobre a aplicação e sobre os passos necessários para

[2] Jair Leite apresenta uma discussão detalhada sobre expressão e conteúdo [Leite, J.C., 1998].

estendê-la. Estes cuidados incluem centralmente um estudo das características comunicativas dessas linguagens.

Inicialmente, considerávamos que uma característica desejável e importante ao se lidar com diferentes linguagens, em uma mesma aplicação, fosse a inter-referencialidade destas linguagens (Draper, 1986). Draper discute a inter-referencialidade de linguagens de entrada e saída, mas utilizamos este conceito para denotar a característica de qualquer linguagem cujos *tokens* se refiram a *tokens* de uma segunda linguagem.

A Engenharia Semiótica, porém, foi além do conceito de inter-referencialidade, e veio a propor como característica importante destas diferentes linguagens, o **contínuo semiótico** entre elas, através dos princípios de Abstração Interpretativa e de Contínuo Semiótico (de Souza, 1999), descritos a seguir. Para tanto, vamos considerar as seguintes definições:

- um **signo** em uma linguagem é qualquer símbolo presente em enunciados desta linguagem, percebido por um ouvinte ou leitor como tendo significado;
- um **signo lexical** é uma palavra desta linguagem;
- um **signo frasal** é um construto gramatical válido desta linguagem, ou seja, uma organização estruturada de signos lexicais que contém um significado completo baseado em modelo;
- um **signo realizado** em uma linguagem é uma palavra ou sentença existente nesta linguagem;
- um **signo potencial** em uma linguagem é uma palavra ou sentença inexistente nesta linguagem, mas que pode ser gerado por extensões em seu vocabulário ou em sua gramática, extensões estas que obedecem os padrões derivacionais desta linguagem.

#### Princípio de Abstração Interpretativa

Dadas duas linguagens computacionais Li e Lj quaisquer, Li é uma **abstração interpretativa** de Lj se:

- a semântica de Li pode ser descrita em Lj

- um usuário de Li pode dar sentido a todos os seus signos lexicais e frasais utilizando três fontes fundamentais:
    i. os padrões de ocorrência destes signos em **discurso situado** em Li, ou seja, sua distribuição e as oposições entre eles;
    ii. algum conhecimento meta-lingüístico extrínseco sobre Li, tal como explicações em linguagem natural sobre Li oferecidas por sua documentação, sistema de ajuda *on-line*, manual de usuário, e outros;
    iii. o conhecimento do usuário sobre computação e sua bagagem cultural.

Em outras palavras, uma linguagem Li é uma abstração interpretativa de uma outra linguagem Lj se, mesmo que a semântica de Li possa ser descrita completamente em Lj, um usuário possa dar sentido a Li sem jamais saber que Lj existe.

### Princípio de Contínuo Semiótico

Dadas duas linguagens computacionais Li e Lj, elas são ditas **semioticamente contínuas** se:

- Li é uma **abstração interpretativa** de Lj
- Lj é uma **abstração interpretativa** de uma outra linguagem Lx
- Lj possui uma definição sintática de **texto** como uma organização estruturada específica de sentenças (cujo significado incorpora elementos intencionais)
- Qualquer usuário que seja completamente proficiente em Li e Lj pode sempre traduzir qualquer texto em Lj em um signo arbitrariamente complexo (realizado ou potencial) em Li.

Em outras palavras, duas linguagens são semioticamente contínuas se uma delas (Lj) for considerada uma linguagem de descrição semântica da outra (Li) e se, além disto, a linguagem de descrição semântica (Lj) possuir um marcador sintático que pode distinguir dois tipos de enunciados pragmaticamente diferentes: os que possuem sentido realizado ou potencial na linguagem descrita (Li), e aqueles que não o possuem. Este componente discrimina a geração de sub-estruturas de sentenças sintaticamente corretas que não fazem sentido na linguagem descrita (Li), mas que fazem sentido na linguagem que a descreve (Lj). Assim, todos os textos com marcação pragmática que ocorrem em uma

linguagem estão conectados a outros da outra linguagem por uma cadeia de significados pretendidos.

Se as linguagens de interface, de extensão e de explicação forem semioticamente contínuas, podemos garantir que uma extensão seja refletida adequadamente tanto na interface quanto no sistema de explicação.

Nosso trabalho estreita ainda mais a relação entre linguagem de interface (UIL) e linguagem de extensão (EUPL), trazendo para a interface mecanismos de extensão que fazem uso dos operadores meta-lingüísticos de metáforas e metonímias. Através da linguagem de interface, o usuário indica o que gostaria de estender, e a aplicação calcula possíveis extensões com base nessa indicação do usuário e, de forma guiada, lhe apresenta estas opções. Este mecanismo será descrito em detalhes no Capítulo 3, "Programação Via Interface".

## 2.3 Engenharia Cognitiva e Aplicações Extensíveis

A Engenharia Cognitiva (Norman, 1986) tem sido usada tradicionalmente como base teórica para estudos de usabilidade de sistemas computacionais. Norman define a distância entre o usuário e a aplicação descrevendo dois golfos: golfo de execução e golfo de avaliação. Para interagir com a aplicação, o usuário precisa atravessar estes dois golfos, através dos seguintes estágios:

- definição do objetivo
- formulação da intenção
- especificação de uma seqüência de ações
- execução da ação
- percepção do estado do sistema
- interpretação do estado
- avaliação do estado do sistema com relação aos objetivos e intenções

A Engenharia Cognitiva apresenta os golfos de execução e de avaliação como sendo elementos estáticos, ou invariantes durante toda a vida da aplicação. Cabe ao *designer* da aplicação original projetá-los estreitos, a fim de facilitar a interação do usuário com a

aplicação. Entretanto, nas aplicações extensíveis, o usuário assume o papel de *designer*, mesmo que de forma limitada, e cabe a ele agora estreitar estes golfos, através de interações com a própria aplicação.

Embora haja afirmações de que uma abordagem centrada no usuário possa dar conta de aplicações extensíveis (Malone, 1995), ao abstrair para fora do modelo o papel do *designer* na criação da aplicação, a Engenharia Cognitiva limita sua aplicabilidade no caso das aplicações extensíveis, pois não oferece previsões claras para o caso de o usuário se tornar ele próprio um *designer*. A Engenharia Semiótica, por sua vez, traz para este cenário o papel do *designer* como sendo responsável pela criação da aplicação e de todas as interações de que ela possa participar. Grande parte da argumentação referente ao *designer* original se aplica também ao usuário–*designer*. A complementação da Engenharia Semiótica para o caso de aplicações extensíveis é a definição, por parte do *designer* original, não apenas das linguagens da aplicação, mas agora também de suas meta-linguagens. As meta-linguagens de uma aplicação extensível delimitam o que usuários finais podem fazer, restringindo as possibilidades de extensão a fim de preservar o significado mínimo da aplicação tal como pretendido pelo *designer* original. Em suma, a limitação da Engenharia Cognitiva está no fato de que se baseia no artefato de software como *produto*, enquanto a Engenharia Semiótica trata do *processo* de desenvolvimento deste artefato, processo este que pode ser continuado pelo usuário, como no caso de aplicações extensíveis.

## 2.4 Representação do Conhecimento

Como mencionado anteriormente, uma aplicação extensível deve possuir como bases de conhecimento um modelo estático e outro dinâmico do domínio, e um modelo da aplicação.

Estas bases de conhecimento devem possuir todas as informações necessárias não apenas para o cálculo de analogias, mas também para produzir as explicações e estender o ambiente, mantendo sob controle as linguagens de representação embutidas na aplicação: linguagens de interface, de programação e de explicação.

O modelo estático do domínio é uma representação da natureza das coisas e de suas *relações estáticas*. Trata-se de uma taxonomia de classes e instâncias de objetos,

acrescida de relações estáticas entre eles. O modelo dinâmico, por sua vez, é uma representação das interações entre as coisas, no tempo, e para um determinado fim. Trata-se de *relações dinâmicas*. O modelo da aplicação contém especificidades do *design* da aplicação e de sua interface, assim como sua interação com o sistema operacional subjacente.

Os modelos são criados pelo *designer* da aplicação. Para contemplar a programação feita por usuários finais, pretendemos dar-lhes recursos para estender estes modelos, de forma controlada. Nossa abordagem quebra barreiras entre interface e ambiente de extensão, incorporando à linguagem de interface os mecanismos metafóricos e metonímicos de extensão, que são mecanismos meta-lingüísticos semântico-pragmáticos, e não apenas lexicais.

Isso nos leva a mais uma questão: Se um modelo do domínio é extensível, como se representa as regras que governam as extensões? E como são representados os novos tipos e operações?

### 2.4.1 Representação dos Modelos do Domínio

Nossa abordagem procura utilizar um sistema de representação de conhecimento com as seguintes propriedades (Rich e Knight, 1994):

- adequação representacional: capacidade de representar todos os tipos de conhecimento necessários ao domínio em questão;
- adequação inferencial: capacidade de manipular as estruturas representacionais de modo a derivar novas estruturas correspondentes, inferindo conhecimento novo a partir do antigo;
- eficiência inferencial: capacidade de incorporar, dentro da estrutura do conhecimento, informações adicionais que possam ser utilizadas para direcionar melhor os mecanismos de inferência;
- eficiência aquisicional: capacidade de adquirir novas informações com facilidade.

Inicialmente procuramos representar os objetos, relações e atributos do modelo estático utilizando apenas predicados do tipo herança (is_a) e composição (part_of). Entretanto, estes predicados não foram suficientes para computar analogias de acordo com os

mecanismos estudados. Foi necessário utilizar uma representação mais rica e estruturada, e embutir na base de conhecimento mais de uma taxonomia.

Como os cálculos de metáforas e metonímias são realizados sobre relações entre os elementos, sejam estes tipos, atributos ou operações, o requisito básico para as linguagens de representação a serem utilizadas é que permitam expressar relações associativas. Tendo em vista esta restrição, deixamos a critério do *designer* escolher a linguagem de representação mais adequada para o seu domínio. Além disto, ele poderá utilizar mais de um tipo de representação, desde que mantenha representada a associação entre os diversos elementos da base de conhecimento. Por exemplo, ele pode utilizar lógica de primeira ordem para representar o modelo estático, e lógica temporal para representar o modelo dinâmico do domínio.

Para o nosso protótipo, escolhemos utilizar uma combinação de métodos declarativos e procedimentais, para poder usufruir das vantagens de ambos. Estes incluem conceitos de gramática de casos (Fillmore, 1968; Bruce, 1975) e dependência conceitual (Schank, 1973; Schank, 1975), que serão descritos no Capítulo 4, "Programação via Interface em Ação".

Para manipular o conhecimento representado, fazemos uso de recursos meta-lingüísticos para calcular extensões à base de conhecimento. Os operadores de extensão serão descritos em detalhes no Capítulo 3, "Programação Via Interface". Por agora, basta saber que os cálculos feitos são de natureza semântico–pragmática, e não apenas lexical. Desta forma, a qualidade das extensões está intimamente ligada ao tipo de conhecimento sobre o domínio que estiver representado, e não à gramática da linguagem de extensão propriamente dita.

Ao trazer alguns recursos de extensão para a linguagem de interface, nosso trabalho oferece um mecanismo atraente para a aquisição de novas informações sobre os elementos e as tarefas de um domínio, a partir da aplicação de operadores de extensão sobre informações existentes.

## 2.5 Analogias

Esta seção apresenta conceitos de analogias e metáforas pelas ciências cognitivas e descreve alguns de seus usos dentro da área de Inteligência Artificial.

### 2.5.1 Metáforas e Metonímias

Nesta subseção apresentamos definições de metáfora e metonímia, descrevemos seu efeito no discurso, e enumeramos algumas distinções feitas por Lakoff e Johnson (Lakoff e Johnson, 1980; Lakoff, 1987; Lakoff, 1993). Através de exemplos, ilustramos possíveis representações que favorecem o uso computacional destes mecanismos para extensão de software.

Tanto metáforas quanto metonímias descrevem uma conexão entre dois elementos, onde um termo é substituído por outro. **Metáforas** são baseadas em semelhança, enquanto **metonímias** expressam relações contíguas simples entre objetos, como parte-pelo-todo, causa-pelo-efeito, etc. (Gibbs, 1993). Em ambos os casos, a significação literal de uma palavra é substituída por outra.

> Um exemplo de metáfora é "*Vejo* o que você quer dizer". Neste caso, *ver* possui um significado não literal: *entender*.
> Um exemplo de metonímia é dizer "Gosto de ler *Machado de Assis*", para se referir às obras deste escritor, e não à pessoa.

O efeito principal da metonímia é **focar** em um aspecto selecionado de um objeto, e esconder os outros, por meio das relações que este objeto possui com outros. Isto é fundamental para as expressões não-literais na comunicação, pois revelam o que é importante para quem fala. Na metáfora, o efeito principal é revelar **pressuposições**. Ou seja, quando alguém se expressa metaforicamente, está dizendo que pressupõe que o ouvinte conheça o campo metafórico e em função dele entenda o dado novo (Brown e Yule, 1983).

Tomamos como ponto de partida os estudos de Lakoff e Johnson, que afirmam que metáforas fazem parte do nosso sistema conceitual, ou seja, que pensamos metaforicamente (Lakoff e Johnson, 1980). A partir de seus resultados, projetamos mecanismos que realizam alguns mapeamentos metafóricos sobre os modelos de uma aplicação. Os estudos de teorias de metáforas (Lakoff, 1987; Lakoff, 1993) nos fornecem eixos universais para tomar como ponto de partida para os cálculos de possíveis analogias sobre os elementos do domínio. Nesta seção descrevemos alguns destes eixos e como podem ser aplicados no nosso modelo.

#### Metáforas de Orientação

Uma maneira de organizar conceitos em estruturas é por meio de metáforas de orientação, principalmente orientação espacial: para cima-para baixo, frente-trás, fundo-raso, central-periférico.

Este tipo de raciocínio pode ser utilizado para navegar através dos modelos do domínio e da aplicação. Pensando em uma representação topológica destes modelos, os deslocamentos de orientação sugeridos podem ser mapeados para: elementos com o mesmo pai, pai-filho, raiz-folha.

#### Metáforas Ontológicas

Uma metáfora ontológica ocorre quando entendemos uma experiência em termos de entidades (ou objetos) e substâncias familiares, e assim podemos agrupar, categorizar e quantificá-las. Eventos e ações são conceitualizados metaforicamente como objetos, atividades como substâncias, e estados como continentes (Lakoff e Johnson, 1980).

Estes grupos e categorizações nos ajudam a perceber relações que não estão óbvias no modelo do domínio, mas que podem ser computadas de forma a apresentar aos usuários finais opções de extensão, como será visto mais adiante, no Capítulo 3, "Programação Via Interface".

#### Metonímias

Uma metonímia procura focar um aspecto selecionado de um objeto, em detrimento a outros aspectos, visando atingir um determinado efeito comunicativo de forma mais simples do que com a expressão literal correspondente. Para tanto, se utiliza de relações do tipo causa–efeito, parte-pelo-todo, local-pelo-evento, produtor-pelo-produto e etc.

Utilizando eixos metonímicos, podemos navegar pela ontologia com base em relações do tipo parte-todo, agente-objeto, agente-instrumento e etc.

### 2.5.2 Analogias e Abdução

Nosso estudo inclui uma investigação sobre raciocínio abdutivo, que pode ser entendido da seguinte forma (Peirce, 1931):

| | |
|---|---|
| regra: | todos os A's que são B's são C's |
| resultado: | este A é C |
| caso: | então, este A é B |

Este tipo de raciocínio difere do raciocínio dedutivo, que parte de regra e caso para o resultado, e do raciocínio indutivo, que parte de caso e resultado à regra. Embora haja imensa controvérsia e discussão sobre a natureza própria da abdução (Hintikka, 1997), a acepção por nós apresentada é uma das possíveis operacionalizações da proposta original de Peirce.

Como o raciocínio abdutivo nos pode ser útil? Outros trabalhos de pesquisa definem o raciocínio abdutivo como a tentativa de gerar novas hipóteses, possivelmente hipóteses explicativas. É um tipo de raciocínio que, ao contrário do raciocínio dedutivo, gera novas informações. Nossos cálculos de analogia fazem exatamente isso: levantam possibilidades de extensão, que podem ser vistas como hipóteses sobre aplicações alternativas.

Outra definição de abdução a trata como um passo de interrogação. De acordo com Isaac Levi (Em Hintikka, 1997):

*"The task of constructing potential answers to a question is the task of abduction in the sense of Peirce."* (*The Fixation of Belief and Its Undoing*, p.71)

*"The 'conclusions' of abductions are conjectures that are potential answers to questions. Deduction elaborates on the implications of assumptions already taken for granted or of conjectures when they are taken, for the sake of argument, to be true. Induction weeds out for rejection some conjectures, leaving the survivors for further consideration."* (ibid, p.77)

Se considerarmos que o procedimento de geração de analogias é uma tentativa de responder à questão: "o que é análogo a X?", podemos traçar um paralelo teórico entre o raciocínio abdutivo e os métodos de cálculo de analogias. No nosso caso, esperamos que o usuário dê uma dica inicial sobre quem é X, ou seja, qual a origem da analogia. A partir deste elemento, vamos percorrendo a base de conhecimento em busca de respostas a esta pergunta, respostas estas que nos indicarão as possíveis analogias que envolvam o elemento selecionado.

Com base na indicação do usuário e nas regras presentes na base de conhecimento, utilizamos um tipo de raciocínio inspirado no raciocínio abdutivo, e que chamamos de **raciocínio com base em explicações**. Este tipo de raciocínio utiliza as associações e classificações dos elementos da base para gerar hipóteses de extensão através de operadores de metáforas e metonímias e, caso necessário, gerar explicações sobre as analogias sugeridas.

### 2.5.3 Metáforas e Semiótica

Outra disciplina que contribui para a compreensão e geração de metáforas é a Semiótica. Umberto Eco faz uso de exemplos literários para mostrar, sob uma abordagem estrutural, que uma metáfora pode ser desmembrada em uma cadeia de conexões metonímicas (Eco, 1979). Em suas palavras, *"the mechanism of metaphor, reduced to that of metonymy, relies on the existence (or the hypothesis of existence) of partial semantic fields that permit two types of metonymic relation: (i) the* codified *metonymic relation inferable from the very structure of the semantic field; (ii) the* codifying *metonymic relation, born when the structure of a semantic field is culturally experienced as deficient and reorganizes itself in order to produce another structure."*

Eco afirma ainda que as conexões entre campos semânticos dentro de um universo são pré-estabelecidas, e que "curto-circuitos metafóricos" são, na verdade, *"uninterrupted web of culturalized contiguity that our hypothetical automaton might be able to traverse through a sequence of binary choices"* (Eco, 1979).

Enquanto Eco descreve um autômato hipotético para criar metáforas, pesquisadores na área de Inteligência Artificial já possuem resultados nesta linha, seguindo cadeias metonímicas para criar analogias em domínios restritos, como será descrito a seguir.

### 2.5.4 Analogias em Inteligência Artificial

As próximas subseções descrevem diferentes estudos de aplicação de analogias na área de Inteligência Artificial: por pressões inter-agentes (French, 1995), por generalização (Furtado, 1992) e por restrições (Holyoak e Thagard, 1996).

Descreveremos mais adiante como extrair do usuário informações sobre possíveis origens de analogia. A partir deste núcleo de analogia, percorremos os modelos do domínio e da

aplicação como num efeito "ripple", no qual vamos nos afastando do núcleo da analogia até, no limite, perder o potencial de cálculo com base semântica ou pragmática.

### Analogias por *Slippages*

Hofstadter e seus colegas do Fluid Analogies Research Group utilizaram micro-domínios para estudar mecanismos humanos de "fluid analogy-making" (Hofstadter et al., 1995).

Metáforas por analogia ou por proporção foram definidas por Aristóteles como sendo do formato A/B=X/Y (Em [Eco, 1986]). O objetivo das aplicações do grupo de Hofstadter é responder a perguntas do tipo "O que é o A de Y?". Afirmar que "X é o A de Y" significa que nos referimos implicitamente ao papel que A exerce com relação a uma entidade B não mencionada mas presumivelmente óbvia.

Na aplicação Tabletop (French, 1995; Hofstadter et al., 1995), por exemplo, há uma lista de "categorias platônicas abstratas". A aplicação lida com pressões mentais, ou **pressões inter-agentes**. Segundo French e Hofstadter, essas pressões nos fazem relaxar as categorizações de um objeto e assim fazer analogias. Segundo eles, "*for a human, the process of perceptually scanning a (tabletop) situation involves focusing briefly on one area and then on another, and having one's attention gradually drawn in more and more to specific areas*". Algumas destas pressões são enumeradas, como por exemplo, pressões de localização, de pertencer a uma categoria ou a grupos. Para defender a aplicabilidade de sua solução a softwares reais e mais complexos, eles enfatizam que estes tipos de pressão são "*universal, or nearly so, in the sense of existing in virtually any domain in which analogies can be made*". Os conceitos utilizados na aplicação Tabletop são apresentados em [French, 1995]. Hofstadter (Hofstadter, 1979) apresenta o conceito de *slippage* como o uso de um conceito no lugar de outro, tolerado mesmo quando não há uma correspondência entre dois conceitos, dentro de um determinado contexto. French expande este conceito, e define três tipos de *slippages*: exportação, transporte e importação.

Um *slippage* de exportação é uma generalização, em uma situação ou *framework*, resultante de uma abstração geralmente no nível básico de categorização (nível básico como definido em [Lakoff, 1987]). Um *slippage* de transporte envolve o mapeamento desta abstração para uma outra situação ou *framework*, reunificando as variáveis livres

obtidas no processo de variabilização. Já um *slippage* de importação ou conceitual consiste na substituição de um ou mais conceitos do esquema original por conceitos adequados que só se aplicam à situação final. Segundo French, a verdadeira geração de analogias envolve os três tipos de *slippages*. Seu trabalho é uma tentativa de fornecer uma "*unified framework in which categorization, recognition, and analogy-making can be seen as being, if not precisely the same, related in a very deep and essential way*" (French, 1995). Seus mecanismos geram analogias a partir de categorizações e informações fornecidas ao computador.

A pesquisa de French também revela a importância da categorização, não apenas de objetos e seus atributos, mas especialmente de possíveis relações entre esses objetos, tais como proximidade, posição externa em um grupo de objetos, grupos de objetos vs. objetos simples, relações estabelecidas previamente, e tipos de composição de objetos, tais como repetição, categoria superior comum e tamanho. Estas relações entre objetos os tornam mais ou menos "destacados", e assim mais ou menos adequados à criação de analogias.

### Analogias por Generalização

Furtado (Furtado, 1992) apresenta um algoritmo para computar analogias através da combinação de cálculos de unificação e generalização. Segundo ele, dois elementos E1 e E2 são análogos se o resultado E3 de sua generalização for não-trivial. Para efetuar este cálculo, ele obtém a generalização mais específica (*msg — most specific generalization*) entre dois termos, que é um termo que retém a informação presente nos dois termos e introduz novas variáveis em caso de conflito.

> Por exemplo, se T1 = p(a,b) e T2 = p(c,b), então msg(T1,T2) = p(X,b).

Enquanto o desafio lançado por Furtado tratava de aplicar este cálculo de analogias a textos narrativos (Furtado, 1992), podemos utilizá-lo sobre os modelos computacionais do domínio e da aplicação.

### Analogias por Tipos de Restrição

Holyoak e Thagard estudam os processos de raciocício analógico sob a perspectiva de uma teoria de restrições múltiplas (Holyoak e Thagard, 1996). Eles descrevem três classes de restrições utilizadas na analogia:

- **semelhança**: raciocínio guiado por uma semelhança direta dos elementos envolvidos
- **estrutura**: raciocínio guiado pela necessidade de identificar paralelos estruturais consistentes entre os papéis nos domínios de origem e de destino das analogias. A principal pressão é para estabelecer um *isomorfismo* entre os elementos de origem e de destino. Em alguns casos, este tipo de restrição é dominante sobre as restrições por semelhança.
- **objetivo**: raciocínio guiado pelos objetivos do raciocinador, ou seja, o que a analogia deve atingir.

Estes três tipos de restrição operam como diferentes pressões no cálculo de analogias, de maneira semelhante às pressões inter-agentes de [French, 1995]. Em outro artigo, Thagard e Verbeurgt descrevem modelos para computar coerência de forma a satisfazer estas restrições (Thagard e Verbeurgt, 1998).

Os algoritmos de cálculos de analogias apresentados nesta seção podem ser utilizados em nosso mecanismo de extensão por metáforas e metonímias, descrito no próximo capítulo.

# 3. Programação Via Interface

Este capítulo descreve um modelo para aplicações extensíveis que torna possível a programação via interface, utilizando os recursos de metáforas e metonímias. Apresentamos as características das aplicações extensíveis geradas a partir do nosso modelo, e descrevemos os mecanismos de cálculo de extensão.

## 3.1 Visão Geral

As aplicações extensíveis atuais disponibilizam principalmente mecanismos de gravação de macros e linguagens de programação (Microsoft, 1994). Outras oferecem ainda mecanismos de programação por demonstração (Cypher, 1993a), algumas das quais incluem mecanismos de inferência para calcular as extensões com base em ações prévias do usuário (Cypher, 1993b; Kurlander e Feiner, 1993; Lieberman, 1993a). No capítulo anterior, na seção 2.1.2, “Problemas e desafios”, indicamos algumas deficiências destas abordagens. Em particular, nosso modelo evita algumas das limitações das gravações de macro, que as tornam pouco úteis fora do contexto específico em que foram gravadas, como ausência de variáveis e de estruturas de controle. Evitamos ainda o excesso de

generalidade e exigência de conhecimento prévio das linguagens de programação, o que dificulta seu uso por usuários leigos; e certas inferências realizadas sem interação com o usuário, que podem fugir de seu controle e de sua compreensão, como em algumas aplicações de programação por demonstração.

Em nossa abordagem, as extensões podem ser feitas de duas maneiras: por interpretação de um enunciado do usuário que não possua sentido literal, ou por construção, através de diálogos do tipo *wizard**. No primeiro caso, os mecanismos de extensão são utilizados para tentar fazer sentido de um enunciado do usuário que não possui interpretação literal. As extensões geradas por este mecanismo são **voláteis**, ou seja, só valem para o caso atual, e não podem ser utilizadas como base para novas extensões. Já para criar uma extensão utilizando *wizards*, um usuário tem que explicitamente pedir para realizar a extensão. O mecanismo de *wizard* o guia passo-a-passo através da extensão e, como resultado, a extensão se torna **persistente**, ou seja, torna-se parte da base de conhecimento da aplicação e fica disponível para ser utilizada em outras extensões.

A distinção entre extensões persistentes e voláteis é necessária para garantir que uma interpretação que valha apenas para uma situação específica não introduza ruído na base de conhecimento. Ou seja, na base deve estar representado conhecimento considerado global, que possa ser aplicado a qualquer contexto. Caso contrário, o mecanismo de extensão passa a gerar alternativas de interpretação e extensão distorcidas, com base em um conhecimento válido apenas para um contexto específico. A necessidade de extensões voláteis também está em linha com estudos de Suchman sobre a natureza situada do aprendizado, memória e compreensão (Suchman, 1987).

É importante notar que o uso dos mecanismos analógicos descritos neste trabalho difere do descrito em [Halasz e Moran, 1982]. Eles consideram analogias que são utilizadas para ensinar aos usuários novos conceitos sobre sistemas computacionais, e alertam para o perigo da utilização de uma analogia em todas as situações. Em nossa abordagem, cabe ao usuário utilizar tais mecanismos para se expressar, ou seja, o usuário é quem "ensina" a aplicação. Além disto, a volatilidade de nosso mecanismo de interpretação evita a "globalização" de uma analogia situada, reduzindo assim o risco de esta analogia ser tomada como válida para todos os casos.

### 3.1.1 Extensão por Interpretação

Os recursos semântico-pragmáticos de metáforas e metonímias podem ser utilizados de forma bastante eficiente na comunicação com a aplicação, principalmente no que diz respeito a casos omissos e em casos de falha de compreensão por parte do usuário.

> Em um editor de textos, quando não há seleção e é acionado o comando de Formatar–Negrito, a aplicação coloca toda a palavra sob o cursor em negrito. Uma interpretação literal nada faria, pois a unidade mínima para esta formatação é o caractere. Este é um exemplo do uso de uma metonímia do tipo continente (palavra) – conteúdo (cursor de texto) (Eco, 1976; Lakoff e Johnson, 1980).

Desta forma, caso haja uma omissão ou ambigüidade na comunicação da intenção do usuário, o mecanismo de extensão procura o elemento mais próximo que seja semelhante ou relacionado ao elemento indicado, ou que se encaixe em seu enunciado.

### 3.1.2 Extensão por Construção via Wizards

No segundo caso de extensão suportado por nosso modelo, os mecanismos de extensão guiam o usuário passo a passo por sua extensão, através de diálogos do tipo *wizard**. O objetivo dos *wizards* é revelar ao usuário as possibilidades de extensão da aplicação, através do potencial de uso de metáforas e metonímias em aplicações computacionais.

Nossos *wizards* apresentam ao usuário diversas opções de extensão, com base em uma indicação inicial sua sobre que tipo ou operação deve ser estendido. Estas opções são calculadas pelos mecanismos de analogia descritos mais adiante, com base nos modelos do domínio e da aplicação. À medida que o usuário percorre os diálogos, os modelos e alguns conceitos de programação lhe são apresentados de maneira casual, e ele pode pedir mais explicações sobre cada passo de extensão. Como efeito secundário deste mecanismo, temos a capacitação gradual do usuário para um nível de extensão que requeira mais conhecimento, como escrever código em uma linguagem de programação propriamente dita.

Muito cuidado deve ser tomado ao apresentar para o usuário as possíveis extensões sobre um trecho de código. Se apresentarmos todas as possíveis opções, corremos o risco de sobrecarregar o usuário com informações excessivas, como pode ocorrer em aplicações extensíveis por configurações de parâmetros (Figura 3).

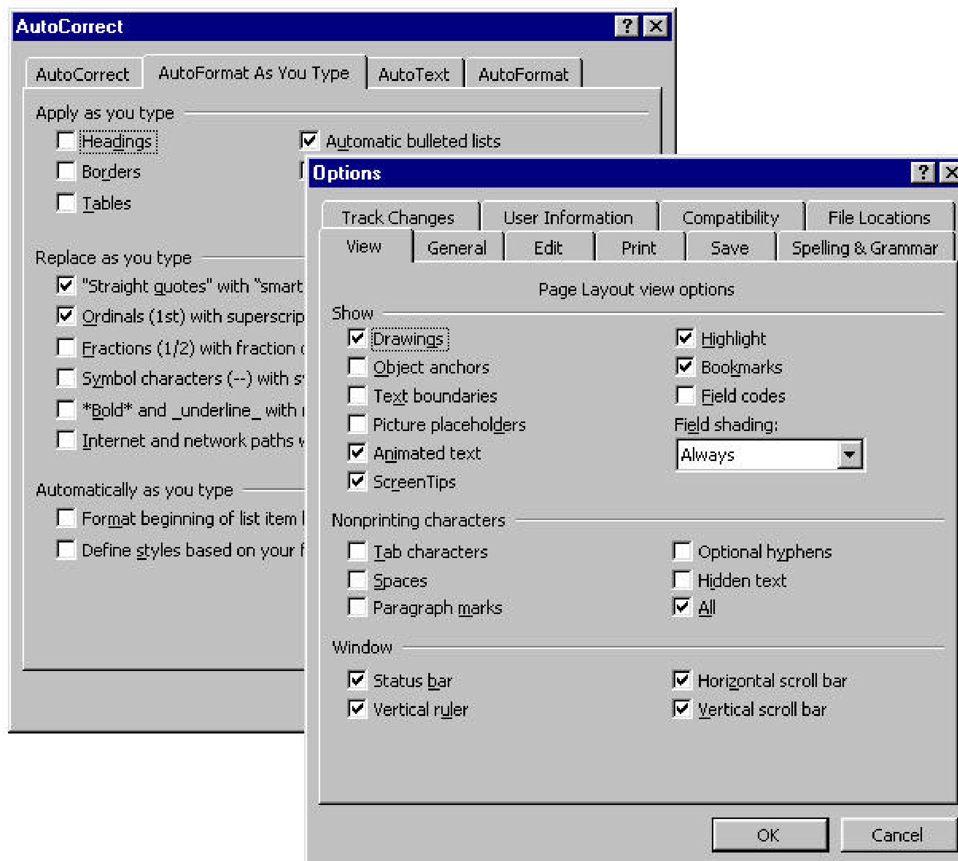


*Figura 3 —Algumas telas de configuração de parâmetros em um editor de textos comercial.*

Nossos recursos de extensão filtram o número de extensões possíveis, procurando apresentar apenas as mais prováveis. Para tanto, utilizamos um princípio de **interpretação local** (Brown e Yule, 1983). Este princípio diz que o ouvinte não deve criar um contexto maior do que o necessário para se chegar a uma interpretação. Em nosso modelo, isto significa calcular os elementos de maior semelhança com o elemento indicado para a extensão, onde apenas um dos elementos enunciados é modificado de cada vez, ou seja, cada extensão sugerida é gerada modificando-se apenas um elemento do enunciado original, tal como projetado pelo *designer* da aplicação. Entretanto, esta restrição pode ser considerada rígida demais para uma determinada aplicação ou uma determinada classe de usuários, ou seja, pode impedir que cálculos interessantes sejam realizados. Para tratar este problema, nosso modelo pode ser estendido para que se possa ampliar o escopo dos operadores de analogia. Este escopo pode ser determinado pelo *designer* como um valor constante ou uma faixa de valores que o usuário pode escolher para regular o potencial de interpretação e geração das metáforas e metonímias. Por

exemplo, podemos navegar pelas cadeias de relações part-of de mais de um elemento envolvido no enunciado, gerando enunciados diferentes.

Em nosso protótipo, a aplicação faz alguns cálculos e sugere as opções que o *designer* original previu como prováveis, mas os usuários têm a possibilidade de pedir que os cálculos sejam realizados com menos restrições, e com escopo mais amplo. Assim, mantemos os usuários no controle de suas extensões, fazendo inferências apenas no sentido de orientá-los, sem no entanto restringir demasiadamente suas tarefas. Para isto, poderíamos definir o conceito de distância entre elementos como sendo o número de relações entre eles. Elementos adjacentes possuiriam distância 1, enquanto elementos entre os quais não houvesse um caminho de relações posssuiriam distância infinita. O usuário poderia então estabelecer a distância máxima a ser navegada pelos mecanismos de extensão, calibrando assim o escopo dos cálculos de metáforas e metonímias.

Um cuidado que devemos ter ao permitir que usuários estendam uma aplicação é não permitir que eles alterem o funcionamento básico da aplicação ou sua interface original. Nossa abordagem impõe uma limitação às extensões possíveis: a aplicação deve vir com um conjunto de recursos que não podem ser revogados ou destruídos pelo usuário, ou seja, as extensões feitas sobre a aplicação possuem caráter **monotônico**. Esta restrição preserva o significado mínimo da aplicação, que representa a "intenção de *design*". Isto é fundamental para manter a consistência do discurso da aplicação, e assim garantir algumas das pré-condições para manter uma comunicação eficiente entre usuário e *designer*, via aplicação. A restrição de monotonicidade também se aplica à interface. Um problema encontrado com freqüência em aplicações extensíveis é a desfiguração da interface após os usuários terem feito suas extensões. Não permitimos, por exemplo, que usuários alterem a operação que é ativada por um item de menu da interface original, como certas aplicações que permitem uma customização total da interface, como o Microsoft Word (Microsoft, 1994) (Figura 4).

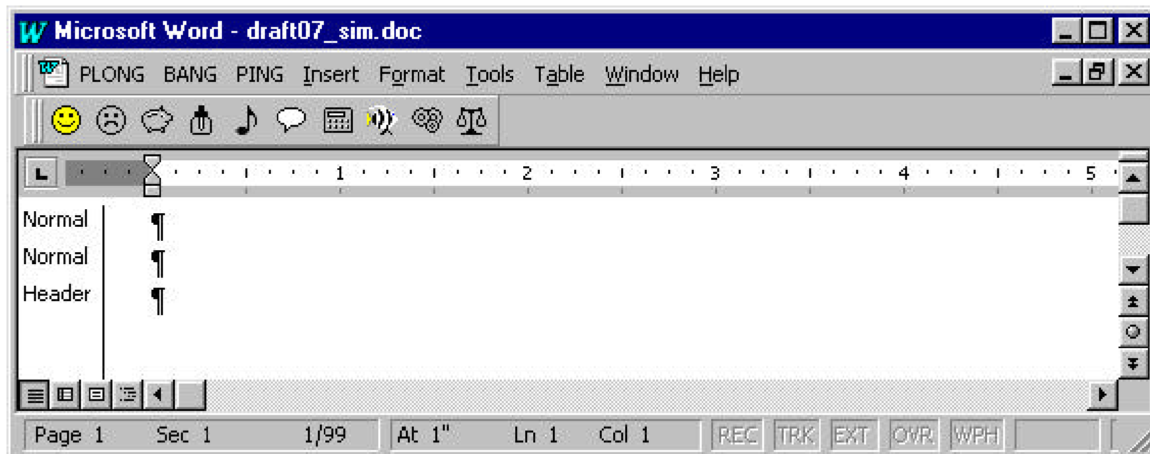


*Figura 4 — Tela de um editor de textos após ter sido customizada pelo usuário.*

> Na tela apresentada acima, os menus File, Edit e View puderam ser substituídos por PLONG, BANG e PING, respectivamente, descaracterizando a aplicação. As barras de ferramenta também foram eliminadas, e outra foi criada em seu lugar, sem que haja qualquer pista sobre o significado de seus ícones.

Como o número de combinações pode ser bastante grande, deve-se classificar ou hierarquizar as alternativas de extensão, de forma a privilegiar algumas em detrimento a outras, garantindo que apenas um número razoável de sugestões de extensão seja apresentado ao usuário. Cabe ao *designer* estabelecer estas classificações, tendo em vista que o usuário poderá, também neste caso, expandir o escopo dos operadores de extensão.

### 3.1.3 Considerações

A nossa abordagem parte de três pressupostos básicos. O primeiro é que pensamos de forma analógica, ou seja, tendemos a pensar em conceitos novos com base em conceitos conhecidos (Lakoff e Johnson, 1980; Lakoff, 1987). O segundo é que, ao interagir com uma aplicação, o usuário tem uma necessidade e está tentando expressar esta necessidade no âmbito da aplicação, a fim de obter como resultado a possibilidade de realização de uma nova tarefa. Estas hipóteses reforçam a nossa opção por fazer de nosso mecanismo de interpretação dos enunciados do usuário um processo de **analogia local** (Brown e Yule, 1983). Desta forma, partimos do enunciado do usuário e, caso não haja interpretação literal para este enunciado, tentamos variar um de seus elementos na tentativa de alcançar uma intepretação válida. A forma como variamos estes elementos é dada pelos operadores de metáforas e metonímias, aplicados aos modelos do domínio e da

aplicação. É importante observar que o usuário não precisa saber o que é uma metáfora ou uma metonímia para utilizar estes recursos em seus enunciados.

Nosso terceiro pressuposto é que, como o usuário forma um modelo conceitual de uma aplicação através de sua interação com ela, dificilmente ele obterá um modelo conceitual correto e completo, o que reforça nossa opção por recursos lingüísticos que permitem que os usuários utilizem enunciados aproximados, muitas vezes incorretos do ponto de vista literal.

Nossos mecanismos permitem interpretar, ou dar sentido a enunciados que não estiverem representados diretamente no modelo, mas apenas através de classificações ou relações com outros elementos. Com isto, um usuário que utilize um modelo conceitual parcial para se expressar de maneira analógica a um enunciado literal terá alguma chance de produzir o resultado esperado, ao invés de receber uma mensagem de erro. Gostaríamos de ressaltar que os cálculos apresentados neste trabalho não pretendem formar um conjunto completo de cálculos possíveis e interessantes, mas apenas um conjunto básico, e ele mesmo extensível, das possibilidades que surgem com tal abordagem.

Nosso trabalho traz alguns mecanismos de extensão para a linguagem de interface, ou UIL (User Interface Language). Dentre as técnicas de programação feita por usuários finais, nossa abordagem se situa entre mecanismos de gravação de macros e linguagens de programação. Possibilitamos que, de forma controlada e através de interações com a linguagem de interface, sejam feitos alguns tipos de extensão na aplicação, utilizando metáforas e metonímias em um raciocínio abdutivo*. Através de associações entre os elementos dos modelos representados na base de conhecimento, conseguimos gerar sugestões de extensões de natureza semântico–pragmática, e não apenas lexical, e que serão aceitas ou descartadas pelos usuários finais. Nosso modelo prevê a existência ou não de uma linguagem de extensão disponível aos usuários finais, ou EUPL (End-User Programming Language). A restrição sobre esta linguagem é que ela seja semioticamente contínua* com a UIL, como descrito na seção 2.2, “Engenharia Semiótica”.

O ponto central do nosso trabalho, entretanto, é quebrar algumas barreiras entre a linguagem de interface (UIL) e a linguagem de extensão (EUPL). Como nosso modelo define na UIL os mecanismos de extensão permitidos, temos uma definição operacional

das estruturas existentes e potenciais que fazem sentido na própria UIL. Com isto, podemos dar sentido a enunciados não-literais bastante interessantes.

## 3.2 Representação do conhecimento

Esta seção apresenta a representação dos elementos necessários aos cálculos de extensão por metáforas e metonímias.

Nossa abordagem permite apenas extensões monotônicas, ou seja, extensões que não permitem destruir ou alterar diretamente os elementos e as relações definidos previamente pelo *designer*, de forma a preservar o significado mínimo da aplicação tal como projetado por ele. Para isto, precisamos distinguir primeiramente o que faz parte deste conjunto mínimo da aplicação, e o que foi acrescentado pelo usuário. Dessa forma, teremos sempre uma marcação nos predicados: os prefixados com a letra “d” são predicados embutidos na aplicação pelo *designer*, enquanto os prefixados com a letra “u” indicam extensões feitas por usuários.

Visto que mecanismos de estender aplicações devem prever não apenas a extensão de sua funcionalidade, mas também da interface correspondente, devemos ainda diferenciar os predicados quanto à sua funcionalidade (ou conteúdo) e quanto à interface (ou expressão). Para isso, prefixamos com a letra “c” predicados relacionados à funcionalidade da aplicação, e com a letra “e” predicados relacionados à interface da mesma.

Dessa forma, temos os seguintes prefixos:

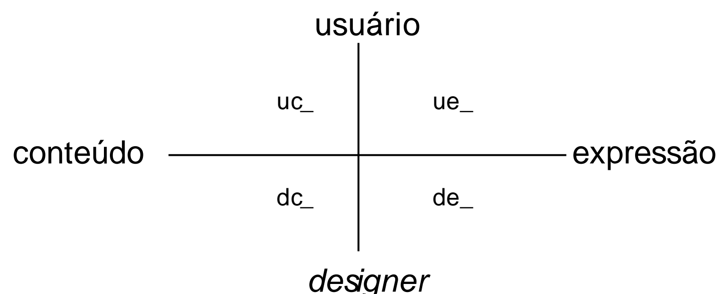


*Figura 5 — Prefixos utilizados na linguagem de representação.*

A tabela a seguir apresenta predicados que ilustram a utilização destes prefixos:

| dc_type(box) | dc_ | tipo de conteúdo definido pelo *designer* |
|---|---|---|
| uc_type(doll) | uc_ | tipo de conteúdo definido pelo usuário |
| de_type(shape) | de_ | tipo expressivo definido pelo *designer* |
| ue_type(composite-shape) | ue_ | tipo expressivo definido pelo usuário |

*Tabela 1 — Ilustração do uso dos prefixos que distinguem designer/usuário, e conteúdo/expressão, na linguagem de representação.*

Estes prefixos são utilizados em toda a representação. Para os predicados que não precisam ou não devem fazer estas distinções, são utilizadas variações, como por exemplo, c_type(Type), que retorna todos os tipos de conteúdo do modelo, tenham sido criados pelo *designer* ou pelo usuário, ou ainda type(Type), que retorna todos os tipos, sejam de conteúdo ou de expressão, criados pelo *designer* ou pelo usuário.

A tabela a seguir ilustra alguns predicados de tipos definidos pelo *designer*:

| **Definição** | **Exemplo** |
|---|---|
| <prefixo>is_a(Tipo, Supertipo) | dc_is_a(robot,actor). |
| <prefixo>part_of(Parte, Coisa) | dc_part_of(block,toy). |
| <prefixo>attribute_of(Propriedade, Tipo) | dc_attribute_of(shape, thing). |
| <prefixo>value_of(Propriedade, Tipo, ValorPropriedade) | dc_value_of(shape, box, box_shaped). |
| <prefixo>relation(Relacao, TipoOrigem, TipoDestino) | dc_relation(is_located,actor, corner). |

*Tabela 2 — Exemplos de predicados sobre tipos definidos pelo designer da aplicação.*

O cálculo de metáforas e metonímias é feito sobre semelhanças e relações entre os elementos da base de conhecimento. Para realizar estes cálculos, o *designer* precisa representar os modelos estático e dinâmico do domínio e da aplicação.

No modelo estático, devem ser representados:

- tipos e classificações dos tipos;
- atributos, dissociados dos tipos, e suas classificações;
- relações estáticas: tipicamente relações de composição, ou relações metonímicas de natureza física, como parte-de e conteúdo–continente.

No modelo dinâmico do domínio, devem ser representados:

- relações dinâmicas e suas classificações: relações genéricas, ou relações metonímicas do tipo causa–efeito, produto–produtor, etc.;
- operações e suas classificações: estas operações devem ser definidas por suas pré- e pós-condições. Devem ser classificadas pelos papéis que os tipos do domínio exercem em cada operação, e quanto ao resultado de cada operação.

## 3.3 Geração de Metáforas e Metonímias

O processo de geração de metáforas e metonímias é essencialmente um processo abdutivo (Peirce, 1931; Hintikka, 1997). Ele levanta possibilidades de extensão, que podem ser vistas como hipóteses sobre aplicações alternativas, e utiliza estas hipóteses para gerar novas informações. No nosso caso, o resultado é uma extensão na funcionalidade e na interface de uma aplicação.

A geração de metáforas é feita sobre semelhanças entre os elementos da base de conhecimento, seja com relação a seus atributos, sua composição ou relações de que participam. A geração de metonímias, por sua vez, é feita sobre determinados tipos de relações entre estes elementos.

### 3.3.1 Eixos de Extensão

Tomamos como ponto de partida de uma extensão um enunciado do usuário, representado por um sintagma da forma sujeito+verbo+complemento, ou variações que omitam o sujeito ou complemento, ou seja, sujeito+verbo e verbo+complemento, respectivamente. Nossos mecanismos de extensão suportam extensão em dois eixos: paradigmático ou sintagmático (Saussure, 1916). Numa extensão no **eixo paradigmático**, criamos um tipo novo com base em um tipo existente, e substituímos o tipo original nos sintagmas que o envolviam originalmente, na tentativa de criar novos enunciados. Uma extensão no **eixo sintagmático**, por sua vez, é uma tentativa de utilizar um tipo existente em um sintagma que não o envolvia originalmente, criando também novos enunciados. Isto significa fazer um uso criativo de uma operação, aplicando-a a um tipo de objeto para a qual não havia sido projetada. De acordo com o princípio de analogia local que seguimos, somente um

elemento do sintagma original pode ser modificado, numa tentativa de preservar ao máximo o significado de cada sintagma, que poderia ser perdido caso fosse permitido variar diversos elementos simultaneamente até, no limite, distorcer completamente o enunciado original.

### 3.3.2 Classificações Utilizadas nos Cálculos de Metonímias

Os cálculos de metonímias são realizados com base em determinados tipos de relações entre os tipos representados nos modelos, tais como: parte–todo, conteúdo–continente, produto–produtor, etc. Embora algumas relações já sejam tidas como metonímias por *default*, como part-of e contained-in, é necessário que o *designer* indique que relações adicionais podem ser utilizadas nos cálculos de metonímias. Ele pode fazer isto utilizando um predicado da forma:

metonymic_relations([created-by, written-by, built-by])

**Exemplo**. Sejam os predicados:
instance_of(karel,robot)
instance_of(lerak, robot)
instance_of(toy1, toy)
is_a(toy,transportable_object)
operation(pick,[robot],[transportable_object])
e a relação
created-by(toy1,lerak)
Utilizando a representação de metonímia descrita acima, posso interpretar pick(karel, lerak) como sendo pick(karel, toy1), visto que a operação pick está definida para transportable_object, e não para robot.

### 3.3.3 Classificações Utilizadas nos Cálculos de Metáforas

Podemos observar, pelos elementos que devem ser representados na base de conhecimento, a presença de diversos tipos de classificação: classificação de tipos, atributos, relações e operações. Estas classificações são fundamentais para a geração de metáforas, que se baseiam na semelhança entre elementos.

Quanto mais rica a classificação dos elementos da base, maiores as chances das inferências feitas pelos cálculos de extensão em conseguir uma boa interpretação para os

enunciados do usuário. No nosso modelo, podemos representar uma classificação de forma extensional ou intensional. Uma classificação pode ser descrita por um predicado da forma:

classification(<nome classificação>, <conjunto de elementos>)

O tipo do elemento classificado pode ser: type, instance, attribute, value, relation, operation. Entretanto, as classificações de types costumam já estar representadas por relações do tipo is_a. Os seguintes predicados ilustram as demais classificações:

```
classification(measurable_stuff, [size, volume, length, width, height, weight])
classification(low_intensity, [small, short, narrow, light])
classification(high_intensity, [large, tall, wide, heavy])
classification(ownership, [have, own])
```

De forma semelhante, nossa base de conhecimento prevê a categorização de ações de acordo com o sugerido em [Schank, 1973; Schank, 1975]. Por exemplo,

```
classification(ptrans, [pick_thing, put_thing])
classification(ptrans, [give, buy, sell, lend, rent])
```

Estes tipos de categorização permitem fazer cálculos de metáforas sobre os elementos da base, como será descrito em detalhes nos mecanismos de extensão apresentados na próxima seção.

## 3.4 Mecanismos de extensão

Esta seção apresenta algoritmos que procuram interpretar os enunciados do usuário ou orientá-lo passo a passo pelas extensões, utilizando diálogos do tipo *wizard*.

### 3.4.1 Tipos de Extensão

#### Extensões Voláteis, ou por Interpretação

Este mecanismo parte do princípio que, ao interagir com a aplicação, o usuário está tentando se comunicar com ela, e para tanto ele transmite enunciados que fazem sentido para ele, e que ele espera que produzam algum resultado na aplicação, visando a realização de uma tarefa. Em aplicações tradicionais, enunciados que não possuem interpretação literal são tidos como erro, tipicamente erros de sintaxe.

Para cada enunciado do usuário que não possua interpretação literal, nosso mecanismo procura dar uma interpretação que produza um resultado na aplicação. Para tanto, é necessário que o enunciado seja um sintagma da forma sujeito+verbo+complemento, e que apenas um elemento não se encaixe nos sintagmas que podem ser gerados pela linguagem projetada originalmente pelo *designer* da aplicação.

> Seja A V B um sintagma definido para a operação V, aplicada ao tipo A e com um complemento do tipo B. Sejam C e D outros tipos. Nosso mecanismo de extensão procura dar uma interpretação aos sintagmas C V B (variação de A para C) e A V D (variação de B para D), mas não para o sintagma C V D (variações de A para C e B para D, simultaneamente). Entretanto, este último caso pode ser gerado pelos de extensão por construção.

Esta restrição se deve ao princípio de **interpretação local**, que afirma que devemos variar o menos possível o contexto do enunciado de forma a tentar fazer sentido dele (Brown e Yule, 1983). Variações concomitantes de A para C e B para D possivelmente conterão interações entre os traços semânticos de D e C, que fogem ao escopo deste trabalho.

A interpretação de um sintagma geralmente é comandada pelo verbo. Um sintagma pode ser representado por um conjunto de tipos que podem assumir o papel de sujeito, e um conjunto de tipos que podem assumir o papel de complemento. As classificações que envolvem estes tipos são utilizadas pelo mecanismo de extensão na tentativa de dar uma interpretação metafórica a um tipo que não pertença ao conjunto de tipos esperados para cada sintagma.

> Sejam A V Z, B V Z, e C V Z sintagmas definidos para a operação V, aplicados ao tipo A, ao tipo B e ao tipo C, respectivamente, com complemento do tipo Z. Sejam também as classificações C1 = {A,D,T} e C2 = {A,B,E,F,T}. Dado um enunciado na forma T V Z, será preciso decidir por uma interpretação de T como semelhante a A, a B, ou a C. É possível que uma ou mais das semelhanças a serem tentadas não leve a nenhuma interpretação consistente (i.e., que o cálculo não seja efetuado). Porém, se todas as opções forem válidas, como A e T estão classificados como semelhantes em pelo menos duas classificações diferentes, a interpretação de T como semelhante a A é preferida. Assim, o sintagma T V Z é interpretado “como sendo” A V Z, e é esta a operação realizada.

O algoritmo utilizado para interpretar um enunciado é:

dado o enunciado E da forma A V B
para cada sintagma S = Z V B definido
  se houver relação metonímica do tipo part-of ou contained entre A e Z, no sentido A–part-of–Z, considere o sintagma Z V B e pare.
  se houver relação metonímica do tipo part-of ou contained entre A e Z, no sentido Z–part-of–A, considere uma iteração, executando a ação correspondente ao sintagma Z V B para todo A e pare.
para cada sintagma S = A V Z definido
  se houver relação metonímica do tipo part-of ou contained entre B e Z, no sentido B–part-of–Z, considere o sintagma A V Z e pare.
  se houver relação metonímica do tipo part-of ou contained entre B e Z, no sentido Z–part-of–B, considere uma iteração, executando a ação correspondente ao sintagma A V Z para todo B e pare.
se houver apenas um sintagma S' = X V B, onde a substituição de X por A resulte em uma operação que possa ser executada, execute a operação correspondente e pare
se houver apenas um sintagma S' = A V X, onde a substituição de X por B resulte em uma operação que possa ser executada, execute a operação correspondente e pare
se houver vários sintagmas S' = Xi V B, que possam ser executados como A V B, então verifique para qual Xi há mais classificações em comum com A, substitua, execute e pare
se houver vários sintagmas S' = A V Xi, que possam ser executados como A V B, então verifique para qual Xi há mais classificações em comum com B, substitua, execute e pare
se não, considere o enunciado A V B e tente criar este sintagma, através de uma extensão persistente na base

No último passo, "tente criar este sintagma" envolve um algoritmo que estende a base através da verificação de um sintagma. Este algoritmo será apresentado na próxima seção.

### Extensão por Construção, Através de Wizards

Este mecanismo parte do princípio que o usuário deseja fazer uma extensão sobre algum elemento existente na aplicação. Isto requer que ele indique este elemento, que chamamos de **elemento de origem da extensão**. Este elemento pode ser um tipo ou uma operação. Para estender um tipo, geralmente o usuário indica uma de suas instâncias, ou *tokens*. Para estender uma operação, ele pode indicar o elemento de interface que a ativa, como um item de menu ou botão na barra de ferramentas. Caso a aplicação possua um mecanismo de acompanhamento do histórico das operações, este também pode ser utilizado para selecionar a indicação de uma operação para extensão. Após indicar o elemento de origem da extensão, ele deve ativar o mecanismo de extensão, utilizando o signo de interface designado para tal.

Ao selecionar um tipo para extensão, nosso mecanismo permite as seguintes extensões:

- Criar um novo tipo de conteúdo (ex: sentenças em um editor de textos): é criada uma duplicata do tipo de origem, que poderá então ser alterada.

- Criar um novo tipo expressivo (ex: estilos de parágrafo em um editor de textos): é criada uma duplicata do tipo expressivo de origem, que poderá então ser alterada.
- Definir o mapeamento conteúdo–expressão de um tipo novo.

Ao selecionar uma operação para extensão, nosso mecanismo permite:

- Criar alguns tipos de iteração, de acordo com a cadeia metonímica de que os tipos envolvidos na operação participam.
- Criar alguns tipos de condicional, segundo os atributos dos tipos envolvidos na operação.
- Definir o mapeamento conteúdo–expressão de uma nova operação, ou seja, acrescentar ou remover um signo de interface para ativar esta operação. Como as extensões são monotônicas, só se pode remover um signo de interface que tenha sido definido pelo usuário.

As próximas subseções descrevem os procedimentos realizados para cada tipo de extensão.

### 3.4.2 Extensões em Tipos

Após indicar um tipo para estender, o mecanismo de extensão cria uma cópia do tipo indicado e gera as seguintes opções, dentre as quais o usuário deverá selecionar uma:

- alterar a composição do tipo, acrescentando partes;
- alterar composição do tipo, removendo partes;
- acrescentar novos atributos;
- acrescentar atributos existentes em um outro tipo;
- remover atributos;
- acrescentar os valores possíveis de um atributo;
- remover valores possíveis de um atributo (e assim criar restrições sobre estes valores).

As extensões que acrescentam partes, atributos, ou valores, podem criar ambigüidades, e as que removem tais elementos podem resultar em operações inválidas, caso haja

operações que façam referência aos elementos que foram removidos. Após realizar uma alteração, o mecanismo de extensão verifica as operações afetadas, a fim de detectar as expressões inválidas ou ambíguas e sugerir novas alterações que as tornem válidas. Este cálculo também é feito com base em metáforas e metonímias. O mecanismo de extensão percorre a base de conhecimento em busca de possíveis alternativas à expressão inválida, de acordo com alguma relação de semelhança entre o elemento indefinido e os elementos a ele relacionados através de atributos em comum, composição semelhante, ou relações em comum com outros elementos.

Caberá ao usuário decidir o que fazer com as expressões inválidas, dentre as opções sugeridas pelo mecanismo de extensão. O procedimento completo é apresentado na próxima subseção. Em linhas gerais, as opções que podem ser sugeridas pelo mecanismo de extensão são:

1. considerar um valor constante;
2. substituir por uma expressão válida;
3. pedir o valor para o usuário, em tempo de execução;
4. criar um novo elemento (parte, atributo, valor ou relação) para substituir um elemento ausente, e utilizá-lo na expressão;
5. eliminar a instrução que envolve a expressão inválida;
6. substituir toda a operação por uma operação semelhante, de acordo com uma classificação na base.

A cada alteração feita sobre um tipo, as operações que envolvem tal tipo são verificadas novamente, para que sejam resolvidas as omissões, ambigüidades e conflitos.

#### Extensão da Base Através da Verificação de um Enunciado

O procedimento abaixo verifica a validade de um enunciado e, caso envolva referências inválidas, tenta estender a base de forma a torná-lo válido.

```
dado um enunciado ou instrução I
se I contiver uma expressão E que não possa ser calculada
    acrescenta_opção_ao_wizard(ignorar instrução I)
    se E envolver um valor V que não faça parte dos valores possíveis do atributo A
        acrescenta_opção_ao_wizard(pedir o valor durante a execução)
        acrescenta_opção_ao_wizard(substituir por uma constante)
        acrescenta_opção_ao_wizard(acrescentar o valor V aos valores possíveis de A, e
                manter a expressão como está)
        se existir uma classificação C para o valor V de A
            se existir outro valor V' de A com a mesma classificação C, e que possa ser
                    utilizado em E
                    acrescenta_opção_ao_wizard(substituir o valor V por V')
        se não
            acrescenta_opção_ao_wizard(substituir o valor V em E por um dos valores
                    possíveis de A)
    se E envolver um atributo A que não faça parte do tipo T
            acrescenta_opção_ao_wizard(acrescentar o atributo A ao tipo T)
            se existir uma relação metonímica entre os tipos S e T, onde A seja atributo de S
                    acrescenta_opção_ao_wizard(utilizar o atributo A de S)
            se existir uma classificação C para A
                    se existir outro atributo A' com a mesma classificação C, que possa ser
                            utilizado em E
                            acrescenta_opção_ao_wizard(substituir o atributo A' por A)
            se não
                    acrescenta_opção_ao_wizard(substituir o atributo A por outro atributo de
                            T)
    se E envolver uma relação R entre o tipo T e outro tipo X, relação esta que não existe
        acrescenta_opção_ao_wizard(criar relação R entre T e X)
        se existir uma relação metonímica entre os tipos S e T, onde R seja relação entre S e
                X
                acrescenta_opção_ao_wizard(utilizar a relação R entre S e X)
        se a relação R possuir uma classificação C
            se existir outra relação R' entre T e X, com a mesma classificação C
                    acrescenta_opção_ao_wizard(substituir relação R por R')
se a instrução I contiver uma expressão E que pode ser calculada
    se o valor de E for sempre constante
            acrescenta_opção_ao_wizard(pedir valor durante execução)
            acrescenta_opção_ao_wizard(substituir pelo valor constante)
            se a instrução I possuir uma classificação C
                se existir outra instrução I' com a mesma classificação C e que possa ser
                    calculada dentro dessa operação
                    acrescenta_opção_ao_wizard(substituir instrução I por I')
    se I for condicional
        se o valor de E for sempre falso
            acrescenta_opção_ao_wizard(eliminar instrução I e o corpo de instruções
                correspondente)
        se o valor de E for sempre verdadeiro
            acrescenta_opção_ao_wizard(eliminar instrução I e mantém o corpo de
                    instruções correspondente)
    se E envolver uma relação R metonímica entre o tipo T e outro tipo X
        se existir uma cadeia metonímica entre X e outro tipo Y
            acrescenta_opção_ao_wizard(cria iteração para todo X de Y)
```

Este procedimento pode ser utilizado para verificar a validade de qualquer elemento que contenha uma seqüência de instruções, seja ele uma operação completa, parte de uma operação, uma pré-condição ou uma pós-condição. No caso de uma operação estar definida através de pré-condições, uma seqüência de instruções e pós-condições, a

verificação desta operação será feita nesta ordem, garantindo que não haja dependências indevidas que possam impedir que o algoritmo pare.

Observe que, em diversos pontos do nosso procedimento, verificamos se existem elementos “com a mesma classificação”. O potencial de aplicação dos operadores metafóricos (baseados nessas classificações de semelhança) é dependente da riqueza da representação dessas destas classificações na base de conhecimento.

> **Exemplo**. Seja size um atributo do tipo block cujos valores possíveis são representados pelo conjunto Values(size,block) = {small, medium, large}. Seja ainda uma classificação Large_qualifiers = {large, tall, wide, huge}. Se o mecanismo de expressão encontrar um *token* block.size.huge, verificará que não é válido, e, em busca de *tokens* válidos com mesma classificação de huge, encontraria o conjunto Large_qualifiers, onde Large_qualifiers ∩ Values(size, block) = {large}, que é a opção sugerida.
>
> Caso não houvesse classificação alguma, os valores sugeridos seriam apenas os elementos de Values(size,block). Se, por outro lado, Large_qualifiers fosse definido de forma incompleta, como Large_qualifiers = {tall, wide, huge}, também não resultaria em nenhuma opção válida.

O procedimento acima considera um tipo simples de classificação dos elementos da base. Para aumentar o potencial das extensões por metáforas, devemos ter diferentes eixos de categorização desses elementos. Desta forma, o procedimento pode ser modificado para que seja levado em conta o número de classificações em comum entre os elementos. Nesse novo procedimento, quanto mais classificações semelhantes houver entre os elementos, maior é sua prioridade dentre as opções de extensão sugeridas ao usuário.

Pode-se limitar o número de elementos semelhantes apresentados ao usuário. Ao atingir um número máximo de elementos, o algoritmo ignora o restante. Entretanto, pode ser interessante permitir que o usuário controle este número máximo de elementos. Assim, ele pode aumentar ou reduzir o alcance dos operadores de extensão, de acordo com sua necessidade. Ele pode fazer isso restringindo os candidatos àqueles que possuírem no mínimo um número m de classificações em comum.

Sejam Set1, Set2, ..., Setn, os conjuntos de elementos classificados como C1, C2, ..., Cn. Podemos definir como melhores candidatos a substituir um elemento e, os elementos pertencentes ao conjunto Best(e,m), onde m é o número mínimo de classificações em comum entre os elementos.
O número de classificações em comum entre o elemento e e um elemento aj é:

$$num(aj) = card \{\forall i \mid e \in Set_i \wedge aj \in Set_i \bullet Ci\}$$

Assim, o conjunto de melhores candidatos a substituir e é:

$$Best(e,m) = \{\forall j \mid num(aj) > m \bullet aj\}$$

Outro refinamento que pode ser feito é organizar as classificações em hierarquias. Dessa forma, pode-se selecionar os eixos de classificação a serem privilegiados nos cálculos de analogias.

Uma classificação pode ser definida extensional ou intensionalmente. Em uma definição extensional, são enumerados os elementos que pertencem à classificação. No caso de uma definição intensional, o conjunto dos elementos que pertencem à classificação é definido por uma regra ou predicado. Uma classificação definida intensionalmente resulta em maior potencial de extensão, visto que a inclusão de novos elementos na classificação pode ser calculada pela regra ou predicado que a define. Em uma definição extensional, entretanto, caberia ao usuário incluir ou não os novos tipos criados em cada classificação. Além disto, um *designer* não consegue antecipar e representar a semântica de uma classificação definida extensionalmente. Neste caso, um usuário poderia acrescentar elementos que não se encaixam na semântica pretendida pelo *designer*, causando distorções semânticas em futuros cálculos de extensão.

Exemplo: Seja M o conjunto de tipos de objetos mensuráveis, ou que possuem tamanho, e I={líquido} o conjunto de tipos imensuráveis. Caso o *designer* defina M={bloco, caixa}, e o usuário crie um novo tipo brinquedo, que também é mensurável, este não é incluído automaticamente em M. Caso o *designer* permita que o usuário insira os novos tipos nas classificações existentes, o usuário deveria explicitamente redefinir M={bloco, caixa, brinquedo}. Entretanto, nada o impediria de redefinir I={líquido,brinquedo}, ou ambos!

Por outro lado, se o *designer* definir $M = \{t \in Types \mid attribute(size,t)\}$, e $I = \{t \in Types \mid \neg attribute(size, t)\}$, então, assim que o usuário criar um novo tipo brinquedo que também possua o atributo size, a definição intensional dos conjuntos garante que brinquedo fará parte de M, e não de I.

### Extensões sobre Tipos Expressivos

No espectro da expressão, somente é possível criar tipos a partir da composição de tipos existentes. Isso ocorre porque os tipos expressivos possuem a informação de como são "desenhados" na tela, e esta informação é fixada no momento de implementação da aplicação. Assim, se a aplicação original definir o tipo círculo, mas não o tipo linha, não se pode criar um tipo triângulo (a não ser, é claro, que se utilize uma degeneração do tipo círculo para criar um tipo ponto, e daí linha; mas este não seria o tipo preferencial de extensão suportado por nosso mecanismo). Fora esta restrição, os cálculos de extensão são feitos como para os tipos de conteúdo, já descritos.

### Mapeamento Conteúdo–Expressão de Tipos

Para cada tipo no espectro do conteúdo, pode haver um tipo expressivo associado. Para os tipos definidos pelo *designer*, não se pode alterar o mapeamento conteúdo–expressão. Já os tipos de conteúdo definidos pelo usuário, através de extensão, devem ser associados aos tipos expressivos correspondentes.

Ao se criar um novo tipo, duplica-se inicialmente o tipo expressivo correspondente ao tipo de conteúdo que deu origem ao novo tipo. À medida que são feitas alterações sobre este tipo, tal como acrescentar e remover atributos ou partes, a estrutura deste tipo é percorrida, de forma a verificar se já existem mapeamentos de suas partes e atributos para elementos no plano expressivo. Além disto, os atributos de cada tipo também podem ser mapeados. Caso haja mapeamentos de conteúdo–expressão dos atributos ou partes do tipo, deverão ser apresentadas ao usuário possíveis "visualizações" do tipo, ou elementos de interface que representam os elementos de conteúdo em questão. Outra possibilidade é apresentar para o usuário um editor gráfico simplificado, onde ele poderia compor os tipos expressivos para refletir seu novo tipo.

## 3.4.3 Extensões em Operações

Quando o usuário indica uma operação para estender, o mecanismo de extensão apresenta as seguintes opções:

1. repetir a operação, criando uma iteração que envolve um ou mais tipos que participam da operação;

2. restringir a operação, criando uma restrição sobre sua execução (uma instrução condicional);
3. capturar o contexto de uma operação, visando a configurar suas pré- e pós-condições.

A seguir descrevemos estas opções.

### Criação de iterações

Para criar uma iteração, o mecanismo de extensão faz uso de operadores metonímicos, percorrendo a estrutura de um tipo e as relações metonímicas entre este tipo e outros, segundo o procedimento a seguir:

```
Para cada instrução I,
    para cada tipo T envolvido em I,
        se existir relação metonímica entre T e U (ex: T part-of U)
            acrescenta_opção_ao_wizard(cria iteração do tipo "para todo T de U")
        para existir cada relação metonímica que exista entre S e T (ex: S part-of T)
            acrescenta_opção_ao_wizard(cria iteração do tipo "para todo S de T")
```

Esse procedimento leva em conta tanto a direção ascendente na cadeia de relações metonímicas (de uma parte para o todo), quanto a direção descendente (do todo para as partes). O mecanismo de criação de iterações deve ser complementado por um mecanismo de criação de condicionais, a fim de restringir a execução das iterações, caso necessário. Este mecanismo é descrito a seguir.

### Restrições na Execução de uma Operação

Para restringir a execução de uma instrução, conjunto de instruções ou de uma iteração, poderíamos criar, através da interface, condicionais, de acordo com o procedimento abaixo:

```
Para cada instrução I,
    se I envolver tipo T
        para cada atributo A de T
            user_show_option(restringir de acordo com algum valor V de atributo A
            de T)
        para cada relação R entre T e outro tipo S
            user_show_option(restringir de acordo com alguma relação envolvendo
            T e S)
```

Entretanto, esta não é uma maneira muito eficiente, pois um tipo pode possuir vários atributos, com diversos valores possíveis, e/ou muitas relações com outros tipos.

### Captura do Contexto de uma Operação

Uma forma mais adequada de restringir uma operação seria configurar pré-condições através de uma indicação do contexto da operação. Para tanto, deveriam ser indicados quais os elementos no contexto anterior à execução da operação são relevantes para sua execução, ou seja, quais os elementos que devem ser considerados preferencialmente no algoritmo acima, para gerar as pré-condições desta operação.

Ao indicar os elementos a serem considerados como contexto de uma operação a ser estendida, estamos na realidade criando classificações temporárias que envolvem estes elementos, e que serão utilizadas no cálculo da extensão atual e descartadas ao final deste processo.

### Mapeamento Conteúdo–Expressão de Operações

Enquanto o mapeamento conteúdo–expressão de tipos trata da visualização de um tipo, o mapeamento de operações trata principalmente dos signos de interface que disparam a operação. Outra distinção feita aqui é quanto ao momento de atualização da visualização das instâncias dos tipos. As instâncias de tipos expressivos podem ser atualizadas a cada passo de execução de uma operação, ou podem ser atualizadas apenas ao final da execução de toda a operação. Esta distinção leva em consideração o que é mais importante na operação: o resultado final (estado final dos elementos de interface) ou o modo como procedimento é efetuado ("trajetória" dos elementos de interface). Caso o *designer* da operação esteja interessado apenas no resultado final, é desnecessário e ineficiente atualizar os *tokens* expressivos a cada passo de execução.

No caso de extensão de operações, geralmente há um ou mais tipos destinados à ativação de operações definidas por usuários finais, como por exemplo, botões em uma barra de ferramentas ou itens de menu. Caso o *designer* deseje aumentar o potencial de extensão da interface, pode permitir que o usuário classifique suas operações de acordo com uma determinada taxonomia, e a partir daí sugerir os tipos e a localização dos elementos de ativação das novas operações. Entretanto, quanto maior for a liberdade do usuário, mais chances há de se romper com o *design* original.

Para restringir as alterações feitas no espectro expressivo da aplicação, o *designer* deve representar na base regras que governam o mapeamento conteúdo–expressão.

> Por exemplo, o *designer* pode representar uma regra que restringe o mapeamento de operações para o tipo expressivo "item de menu". Desta forma, somente este tipo de signo de interface poderá ser utilizado para acionar operações através da interface. Neste caso, uma operação estendida não poderia ser disparada por um botão de comando, por exemplo.

Para tanto, é necessário categorizar os signos de interface de maneira a refletir os sintagmas válidos na linguagem definida pelo *designer* original. Uma extensão em uma operação resultará em um novo sintagma, cujo signo de interface terá que satisfazer as regras estipuladas para os sintagmas originais. Como os sintagmas estendidos são gerados a partir de uma alteração em apenas um dos componentes de um sintagma existente, uma interface bem estruturada facilitará a definição e o posicionamento dos novos signos de interface que deverão ser criados para refletir a extensão de funcionalidade.

> Considere uma hierarquia de menus organizada da seguinte forma: menu ↔ operação, submenu ↔ elemento a que a operação é aplicada; sub-submenu ↔ complementos. Seja o mapeamento de um sintagma A V B para o item V–A–B na estrutura de menus. Caso seja criado um novo sintagma A' V B, este deverá ser mapeado para um item V–A'–B, ou seja, será criado um novo submenu A' do menu V, com um sub-submenu B. Analogamente, um novo sintagma A V B' deverá ser mapeado para um item V–A–B', ou seja, um novo item B' no sub-submenu V–A.

Estes mapeamentos devem ser especificados completamente pelo *designer* e representados na base de conhecimento. Assim, as extensões feitas na funcionalidade da aplicação poderão ser refletidas de forma controlada na interface, a fim de manter a consistência com o *design* original.

## 3.5 *Wizards*

Este capítulo descreveu até agora como as possíveis extensões são calculadas com base em analogias. Esta seção descreve como as opções de extensão devem ser apresentadas aos usuários, no caso das extensões persistentes ou geradas por construção, através de diálogos do tipo *wizard*.

Do ponto de vista do usuário, as opções de extensão lhe serão apresentadas de forma casual, com a possibilidade de acessar explicações sobre como estas opções foram calculadas. No caso de uma extensão por construção, mesmo que apenas uma opção seja

encontrada, esta será apresentada para o usuário. Isto é importante para o usuário se sentir no controle da aplicação, e saber que ele sempre poderá rejeitar as opções que lhe forem oferecidas. Este ponto distingue claramente nossa abordagem daquelas onde não se presume que haja falhas na geração de analogias e o algoritmo é executado automaticamente, como por exemplo, mecanismos de automatização de edição e formatação de textos, habilitados por *default* em alguns editores de texto comerciais.

Em nossa abordagem, utilizamos um mecanismo semelhante ao diálogo através de *wizards*, que guiam o usuário passo a passo através das etapas de extensão. Nossos *wizards* apresentam *widgets* (componentes visuais interativos) familiares ao usuário, tais como: campos de texto, botões de seleção (*check boxes*), botões de opção (*radio buttons*), listas com seleção simples ou múltipla e botões de comando. Estes *widgets* também possuem legendas, que serão utilizadas para descrever as opções geradas de acordo com a linguagem de explicação projetada no ambiente.

O acesso seqüencial via *wizards* visa a reduzir a carga cognitiva sobre o usuário da diversidade de opções disponíveis, estruturando-as de acordo com regras que permitem a usuários freqüentes prever o próximo passo. Idealmente, em termos do modelo de interação, a cada momento, o usuário poderá ver o que está sendo gerado de diferentes maneiras, visualizando o modelo estático, o modelo dinâmico, vendo o código textual gerado (caso haja uma linguagem de extensão textual disponível ao usuário final) ou pedindo explicações sobre o passo atual. Entendemos que, desta forma, revelamos gradualmente aos usuários conceitos da aplicação e de programação em geral. Assim, os usuários mais interessados podem aprender com a aplicação para, caso necessário, passar a escrever código diretamente, sem o auxílio dos cálculos de analogia e da apresentação passo a passo através de *wizards*.

## 3.6 Discussão

Os procedimentos de extensão definidos nas seções anteriores apresentam algumas questões que devem ser esclarecidas sob a luz das teorias em que nosso trabalho se baseia. Uma primeira questão importante se refere às classificações de que os mecanismos de extensão fazem uso: Como saber que tipo de classificação deve ser representada?

Os algoritmos apresentados se baseiam nas classificações dos elementos representados na base de conhecimento. Se tivermos um esquema de classificações rico, os procedimentos geram extensões melhores, ou seja, que fazem mais sentido de acordo com a semântica embutida na base de conhecimento. Quanto mais semântica for representada na base, por meio das categorizações, mais próximos os mecanismos se tornarão dos mecanismos de metáforas e metonímias em linguagem natural.

Se, por outro lado, tivermos um esquema de classificação mais simples, as extensões geradas terão uma relação mais fraca com a semântica do domínio e da aplicação. No limite, quando não houver classificação alguma, os procedimentos trabalharão com associações de natureza sintática apenas, pouco fazendo além de computar interpretações literais dos enunciados.

> Por exemplo, seja um atributo size com valores value_of(small,size), value_of(medium,size), value_of(large,size). Seja também um *token* size.big que precisa ser interpretado. Caso haja uma classificação: classification(big_things, [big,large,long]), obterei como opção de interpretação a substituição de big por large. Entretanto, se não houver uma classificação que envolva big e um dos valores válidos do atributo size, as opções de interpretação seriam de natureza sintática, ou seja,
> a) substituir big por um dos valores válidos de size, dentre [small, medium, large];
> b) substituir big pelo valor do atributo size da instância ...
> b) pedir ao usuário o valor de big em tempo de execução; ou
> c) considerar size.big constante e igual a...

Estes mecanismos são limitados, se comparados aos mecanismos presentes na linguagem natural. Em particular, são muito limitados se aplicados a linguagens que possuem um nível baixo de articulação, como linguagens de interface inteiramente visuais, de manipulação direta ou icônicas.

A capacidade do ser humano de fazer associações metafóricas ou metonímicas está vinculada a nossos esquemas de categorização. Lakoff descreve alguns níveis de categorizações, onde as de nível básico são aquelas caracterizadas principalmente por percepção gestáltica, capacidade de visualização, interação motora, função social e memorização (Lakoff, 1987). É neste nível de classificação que nos expressamos com mais freqüência. Portanto, é desejável que o *designer* represente os elementos do domínio e da aplicação primeiramente neste nível, recorrendo aos níveis mais abstratos ou mais especializados de acordo com a necessidade, segundo uma opção consciente de suas

implicações. Uma classificação em um nível mais abstrato implica menos distinções, e conseqüentemente menor potencial de extensão. Por outro lado, uma classificação em níveis mais especializados pode produzir mais detalhes do que se pretende. Classificações muito complexas podem exigir uma interação maior com o usuário, para decidir qual das muitas alternativas geradas corresponde à extensão desejada. De qualquer forma, esta decisão deve ser tomada pelo *designer*, de acordo com o domínio e o modelo da aplicação, tendo em vista a classe de usuários-alvo.

> **Exemplo**. Sejam os tipos robot, block, toy, e beeper. Consideremos os três grupos de classificação a seguir:
>
> 1) Stuff = {robot, block, toy, beeper}
> 2) Actors = {robot}, Things = {block, toy, beeper}
> 3) Actors_with_wheels_for_legs = {robot},
>    Things_that_are_solid_and_regular_shaped = {block},
>    Things_that_are_made_up_of_blocks = {toy},
>    Things_that_beep_3_times = {beeper}
>
> De acordo com os procedimentos de extensão descritos acima, para cada tipo inválido verificamos se há tipos com mesma classificação que possam substituí-los. Neste exemplo, podemos observar que, no nível de classificação mais abstrato (1), não conseguimos fazer distinção entre os elementos. Por outro lado, o nível minuciosamente mais especializado (3) também não é muito útil, pois nenhuma classificação contém mais do que um elemento. Através deste exemplo, ilustramos que é necessário um equilíbrio, e que este equilíbrio só pode ser obtido através de uma análise expressiva (ou semiótica) do domínio e dos elementos que o *designer* deseja disponibilizar para as extensões na aplicação.

O potencial de extensão de uma aplicação pode ser ainda maior caso o *designer* disponibilize mecanismos para estender as próprias classificações, dando ao usuário poder de criar semântica. Entretanto, estas extensões não podem ser quaisquer. Considerando uma classificação como um conjunto definido intensionalmente, deve-se restringir as operações sobre conjuntos que possam gerar novas classificações, a fim de manter o contínuo semiótico* da aplicação resultante. Caso fosse possível utilizar os operadores lógicos de conjunção, disjunção e negação, o usuário poderia criar novas classificações que negassem a semântica pretendida pelo *designer* da aplicação, rompendo com o nosso requisito de monotonicidade das extensões.

Há casos em que o *designer* da aplicação pode querer disponibilizar qualquer tipo de extensão sobre as classificações. Tais aplicações seriam tipicamente ferramentas de *design*, onde os usuários seriam eles próprios *designers* que precisariam de muita liberdade, mas que também teriam todo o conhecimento necessário para não criar uma aplicação com distorções semânticas.

Precisamos então definir que operações sobre os conjuntos podemos disponibilizar, de forma a permitir acrescentar semântica à base de conhecimento, sem no entanto entrar em conflito ou negar a semântica previamente definida. As operações de união, interseção e diferença de conjuntos podem ser utilizadas com segurança, mantendo a monotonicidade das extensões, visto que não podem anular as informações presentes na base.

### 3.6.1 União de classificações existentes

Sejam C1 e C2 duas classificações distintas. A classificação $C = C1 \cup C2$ resulta em uma classificação em um nível mais abstrato, obtida ao se *relaxar* um ou mais traços semânticos.

> **Exemplo**: Seja {pequeno, médio, grande} o conjunto de valores possíveis do atributo tamanho. Se $C1=\{x \in \text{Types} \mid x.tamanho=pequeno \wedge f(x)\}$ e $C2=\{x \in \text{Types} \mid \neg(x.tamanho=pequeno) \wedge f(x)\}$, então $C = C1 \cup C2$ resulta em uma classificação dos elementos $C = \{x \in \text{Types} \mid f(x)\}$, onde o tamanho não é relevante.

### 3.6.2 Interseção de classificações existentes

Sejam C1 e C2 duas classificações distintas. A classificação $C = C1 \cap C2$ resulta em uma classificação em um nível mais específico, obtida ao se restringir os elementos com relação a um ou mais traços semânticos.

> **Exemplo**: Seja {pequeno, médio, grande} o conjunto de valores possíveis do atributo tamanho. Se $C1=\{x \in \text{Types} \mid \neg (x.tamanho = pequeno)\}$ e $C2=\{x \in \text{Types} \mid \neg (x.tamanho = grande)\}$, então $C = C1 \cap C2$ resulta em uma classificação dos elementos $\{x \in \text{Types} \mid x.tamanho = médio\}$.

Se $C1 \cap C2 = \emptyset$, este tipo de extensão não fará efeito sobre a base ou sobre os futuros cálculos de extensão de tipos e operações.

### 3.6.3 Diferença entre classificações existentes

Sejam C1 e C2 duas classificações distintas. A classificação C = C1 – C2 resulta em uma classificação paralela, que pode resultar em uma especialização de C1 com relação aos traços semânticos relevantes a C2.

> **Exemplo**: Seja {pequeno, médio, grande} o conjunto de valores possíveis do atributo tamanho, e {leve, pesado} o conjunto de valores possíveis do atributo peso. Se C1={x ∈ Types | x.tamanho = grande} e C2={x ∈ Types | x.peso = pesado }, então C = C1–C2 resulta em uma classificação dos elementos {x ∈ Types | x.tamanho = grande ∧ ¬ (x.peso = pesado)}.

### 3.6.4 Especialização de uma classificação existente

Seja C uma classificação. A classificação C1 resulta da especialização de C, com relação a um traço semântico, e dá origem também a $C1^{-1}$, seu conjunto complementar com relação a C.

> **Exemplo**: Seja C = {t ∈ Types | f(t)}. Criamos um subconjunto C1 = {t ∈ C | t.tamanho = pequeno}, e por conseguinte $C1^{-1}$ = C – C1 = {t ∈ C | ¬ (t.tamanho = pequeno)}.

Às extensões apresentadas acima, podemos associar expressões lingüísticas, visando a apresentá-las de forma casual aos usuários finais, que podem não ter noções de operações sobre conjuntos. Por exemplo, sejam X e Y os nomes de classificações distintas: a união poderia ser expressa por "tudo que for X **ou que for (também)** Y"; a interseção por "tudo que for X **e também for** Y"; a diferença poderia ser expressa por "tudo que for X **mas não** Y"; e o complementar, **tudo que não for** X.

Também é interessante notar que, para cada classificação C de um elemento E, existe uma relação da forma classified_as(E,C), relação esta que deve lidar não com atributos, mas sim traços semânticos, ou *differentiæ* (Eco, 1976). Estas classificações definem uma nova rede de relações sobre os modelos do domínio e da aplicação. Podemos utilizar a noção de perspectiva em que determinada classificação se aplica, como em classified_as(E,C,P) (Oliveira, 1999). Cada perspectiva demarca uma espécie de leitura específica do domínio, impedindo argumentações e expressões que não façam sentido em determinadas situações de uso. Neste caso, o modelo do domínio pode ser visto como uma macro-perspectiva, ou uma perspectiva global, que define os elementos e relações invariantes, ou seja, válidos em todas as perspectivas.

# 4. Programação via Interface em Ação

Este capítulo descreve o uso do nosso modelo, apresentando alguns tipos de representação e classificação que podem ser vistos como orientações para um *designer* que queira utilizá-lo. Apresentamos nosso protótipo, e ilustramos alguns caminhos de interação com os mecanismos de *wizards* para estendê-lo. Pretendemos assim ilustrar como o estudo realizado pode ser utilizado no *design* de outras aplicações extensíveis, e aplicado a outros domínios.

## 4.1 O Protótipo

Nosso protótipo foi inspirado no robô Karel, utilizado por Pattis para ensinar conceitos básicos de programação (Pattis, 1995; Bergin et al., 1997). O nosso modelo consiste em um mundo bi-dimensional, no qual um robô pode se mover em um *grid*, e interagir com objetos do mundo. A Figura 6 apresenta uma porção de tela do nosso protótipo, ilustrando

o robô na esquina (4,3) e olhando para o leste, dois beepers nas esquinas (3,4) e (6,4), representados pelos círculos em vermelho, e dois muros, representados em azul.

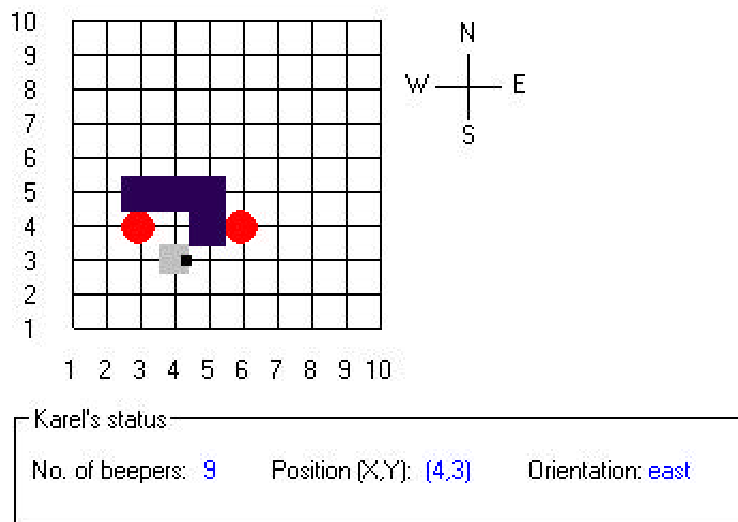


*Figura 6 — Porção de tela ilustrando o mundo de Karel.*

A Figura 7 apresenta uma porção do modelo estático do domínio do protótipo.

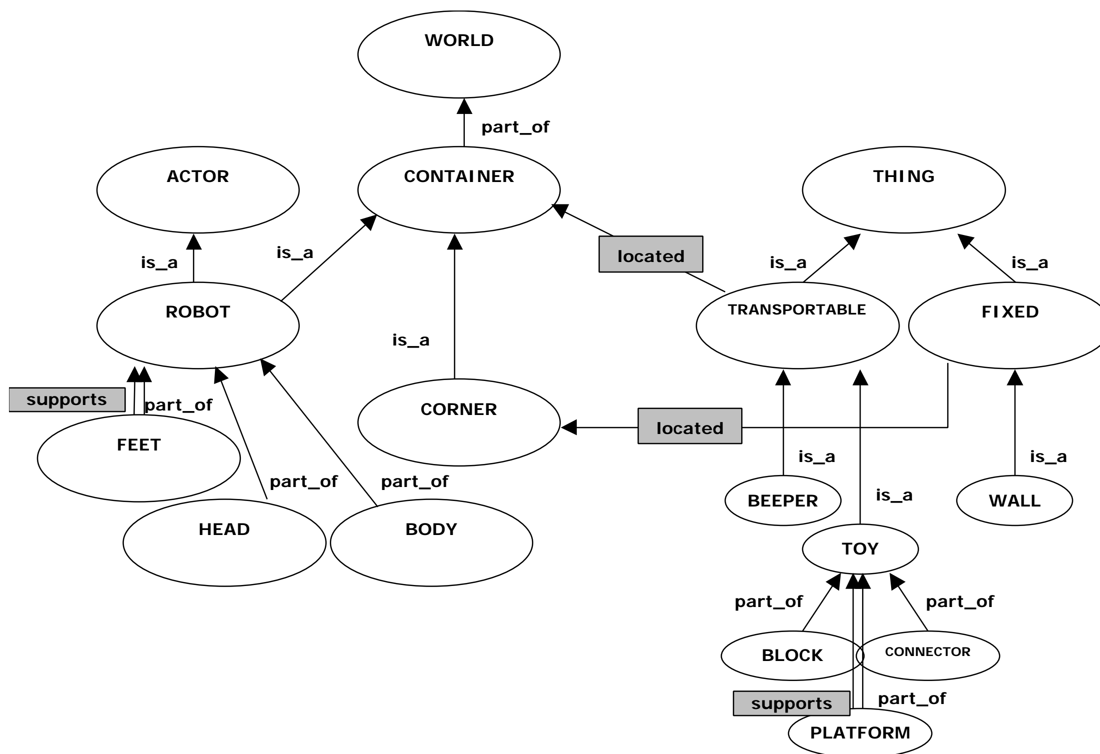


*Figura 7 — Representação de parte do modelo estático do domínio do nosso protótipo.*

As próximas seções descrevem alguns tipos de classificações que podem ser utilizados nos cálculos de extensão. Sempre que possível, são apresentados exemplos com base em nosso protótipo.

## 4.2 Modelo Estático

Para classificar os elementos do modelo estático, utilizamos categorizações que com freqüência geram metáforas, tal como mostrado por Lakoff e Johnson (Lakoff e Johnson, 1980). São as metáforas de orientação, personificação e de condutor.

### Personificação

Com freqüência utilizamos características humanas para descrever aspectos de objetos físicos, ou mesmo abstratos. Exemplos destas classificações são:

```
classification(things_on_top, [top,head])
classification(things_in_middle, [middle,body])
classification(things_at_bottom, [bottom,foot])
classification(personification, [person, robot, toy])
```

Este tipo de classificação pode ser utilizado para permitir que objetos assumam o papel de agentes em algumas operações do tipo agente+verbo+objeto, ou vice-versa.

### Metáforas de Condutor

Estas metáforas são utilizadas para classificar elementos que podem ser considerados objetos, que são então colocados em continentes e transmitidos por um condutor até um destinatário.

```
classification(conduit_object,[idea, letter, person, beeper])
classification(conduit_container,[word, envelope, car, box, robot])
classification(conduit_conduit,[air, postal service, tunnel, street, avenue])
```

### Metáforas de Orientação

Com base em nossa experiência física e cultural, alguns conceitos podem ser representados com relação a uma orientação espacial:

```
classification(orientation_up,[more,up,good,high,north,bigger,better,happy,previous])
classification(orientation_down, [less,down,bad,low,south,smaller,worse,sad,next])
```

> Neste caso, um enunciado "go up" poderia ser interpretado como "go north". Esta classificação é arbitrária, convencionada pelo *designer*. Em outra representação, poderíamos ter uma semelhança entre up e south, por exemplo.

> Em um outro exemplo, um editor de textos que utilize a metáfora de livro classifica como semelhantes left e previous em alguns países, ou right e previous, em outros.

Estas metáforas podem ser utilizadas pelo *designer* para representar os mapeamentos conteúdo–expressão.

## 4.3 Modelo Dinâmico

Para representar a classificação dos elementos no modelo dinâmico do domínio, em nosso protótipo, utilizamos conceitos de gramática de casos (Fillmore, 1968; Bruce, 1975) e dependência conceitual (Schank, 1973; Schank, 1975). Para utilização no cálculo de metáforas e metonímias, uma operação é representada no nosso modelo pelos tipos envolvidos na operação, seus papéis, e as pré- e pós-condições da operação. Não objetivamos aqui advogar em favor desta ou aquela representação de conhecimento, mas apenas ilustrar o uso de algumas representações como possíveis formas de modelar o domínio a fim de possibilitar a geração de metáforas e metonímias.

### Metáforas de Relações Causais

As metáforas de relações causais podem ser úteis na representação de uma operação, que envolve agentes, objetos, instrumentos, estado inicial e estado final. Elas utilizam tanto os papéis que os tipos assumem ao participar da operação, quanto os resultados da execução da operação.

Para representar estas classificações, utilizamos conceitos derivados de gramáticas de casos, e de redes de dependência conceitual. Mais especificamente, utilizamos os conceitos derivados de gramáticas de casos para classificar os tipos referenciados por uma operação de acordo com os papéis que exercem nesta operação, e os conceitos de ações primitivas de Schank (Schank, 1973; Schank, 1975) para representar o resultado ou efeito de uma operação.

### Gramática de Casos

Para representar os papéis exercidos pelos tipos nas operações do modelo dinâmico do nosso protótipo, podemos utilizar os seguintes conceitos:

- A (agente): instigador da ação (tipicamente animado)
- I (instrumento): causa do evento ou objeto usado para causar o evento (tipicamente inanimado)
- D (dativo): entidade afetada pela ação (tipicamente animado)
- F (factitivo): objeto ou ser resultante do evento
- L (locativo): local do evento
- S (fonte): local a partir de onde algo se move
- G (meta): local para onde alguma coisa se move
- B (beneficiário): ser em cujo interesse ou benefício o evento ocorreu (tipicamente animado)
- O (objeto): entidade que recebe a ação ou que se transforma, o caso mais genérico

Os predicados a seguir ilustram a classificação de algumas operações válidas no domínio do nosso protótipo.

```
op_classification(role, [move], agent, [robot])
op_classification(role, [pick, put], agent,[robot])
op_classification(role, [pick, put], object, [transportable_object])
op_classification(role, [build, assemble, disassemble], agent, [robot])
op_classification(role, [build, assemble, disassemble], instrument, [tool])
op_classification(role, [build, assemble], object, [block connector])
op_classification(role, [build, assemble], factive, [toy])
op_classification(role, [disassemble], object, [toy])
op_classification(role, [disassemble], factive, [block, connector])
```

### Dependência Conceitual

Reproduzimos aqui um conjunto de ações primitivas retirados das redes de dependência conceitual que julgamos úteis para classificar o modelo dinâmico do domínio do nosso protótipo.

| ATRANS | transferência de uma relação abstrata | dar |
|---|---|---|
| PTRANS | transferência da localização física de um objeto | ir |
| PROPEL | aplicação de força física a um objeto | empurrar |
| MOVE | movimento de uma parte do corpo de alguém | levantar-se |
| GRASP | segurar um objeto | agarrar |
| BUILD | construir novo objeto a partir de um objeto existente | montar |

```
op_classification(result, [move], [ptrans])
op_classification(result, [pick, put], [atrans])
op_classification(result, [build], [build])
```

É importante observar que estas classificações são dependentes de domínio. As classificações apresentadas aqui são úteis para o domínio do nosso protótipo, no qual um robô se move em um plano, e pode carregar objetos ou depositá-los no mundo. Outros domínios requerem outros conjuntos de classificações. Cabe ao *designer* utilizar os recursos de classificação mais adequados ao domínio de sua aplicação.

## 4.4 Mapeamentos Expressão–Conteúdo

Após estender a funcionalidade da aplicação, precisamos ainda estender a interface para refletir as alterações feitas. Isso levanta as seguintes questões:

Como ligar um trecho de código a um signo de interface? É possível manter uma correspondência entre trechos de programação e signos de interface? Em outras palavras, podemos identificar padrões de programação correspondentes a certos signos de interface? Responder estas questões é um passo essencial na solução do problema de manter a correspondência entre a linguagem de interface e a linguagem de extensão.

Nossa abordagem trata essas questões de maneira análoga à extensão da funcionalidade. Representamos na base de conhecimento os tipos expressivos, relações entre eles e, principalmente, mapeamentos que o *designer* previu entre conteúdo e expressão, ou seja, entre elementos de funcionalidade e de interface.

Da mesma forma como na extensão da funcionalidade, aqui também as extensões devem ser monotônicas, preservando o significado mínimo da interface original. Mas há limitações adicionais aqui, pois o usuário não pode incorporar primitivas de expressão que não tenham sido previstas pelo *designer*. Por exemplo, se o *designer* original não tiver previsto o uso de cor, não há nenhuma função da aplicação que trate esse atributo, e portanto o usuário não teria como utilizá-lo. Por outro lado, caso o *designer* tenha oferecido um conjunto abrangente de primitivas, os usuários terão bastante liberdade para fazer composições dessas primitivas, resultando nos mais diversos tipos expressivos.

A tabela a seguir ilustra alguns mapeamentos conteúdo–expressão representados em nosso protótipo:

| CONTEÚDO | DESCRIÇÃO DA EXPRESSÃO |
|---|---|
| **posição** | **coordenada** |
| avenida X, rua Y | (x,y)=mundo_para_tela(X,Y) |
| **forma** | **representação 2D** |
| esfera (sphere) | pequeno círculo com padrão sólido |
| forma de muro (wall-shaped) | grande quadrado com padrão em diagonal |
| cubo (cube) | pequeno quadrado com padrão sólido |
| caixa (box-shaped) | grande quadrado oco |
| **orientação** | **Forma com orientação ou "olho" no quadrado que a circunscreve** |
| N [S, L, O] | forma com ponto de orientação apontando para o norte [sul, leste, oeste] ou "olho" no topo [parte inferior, lado direito, lado esquerdo] do quadrado que a circunscreve |

*Tabela 3 — Exemplos de mapeamento conteúdo–expressão representados em nosso protótipo. Os mapeamentos estão agrupados de acordo com os valores de cada atributo.*

Assim, ao se estender um tipo de objeto acrescentando-se um ou mais atributos, a aplicação deve verificar se algum mapeamento deve ser obedecido. Quanto mais gramaticalizado for este mapeamento, mais fácil será diagnosticar as alterações e resolver os conflitos que podem surgir da combinação de diferentes tipos de atributos.

> Por exemplo, vamos criar um novo tipo de objeto chamado blip, que deve ser uma espécie de beeper (objeto sem orientação cuja representação é uma esfera), mas com o atributo orientation. Esse novo atributo deve ser refletido na expressão do objeto, nesse caso em sua representação na interface. Examinando os mapeamentos de atributos de orientação, vemos que existem duas opções: ter um ponto de referência para a orientação (acrescentando um elemento gráfico adicional, como um nariz), ou indicar por um quadrado circunscrevendo o objeto, com um ponto indicando a orientação. Cabe ao usuário decidir, mas as restrições de representação impostas pelo *designer* devem ser mantidas, visando manter o significado da aplicação.

De maneira semelhante à extensão de tipos, uma extensão de funcionalidade ou operação também deve ser refletida na interface. Para tanto, utilizamos os *widgets* (componentes visuais interativos) que estiverem disponíveis no ambiente computacional utilizado. A tabela a seguir apresenta um conjunto típico de *widgets*, presentes em grande parte dos ambientes gráficos atuais (Apple, 1992; Microsoft, 1995b). Estes widgets são utilizados

no mapeamento de operações, tanto com relação à sua ativação quanto à entrada de dados para os parâmetros (complementos) destas operações.

| widget | comportamento interativo | comportamento funcional |
|---|---|---|
| botão de comando (pushbutton) | parece "pressionado" enquanto o botão do mouse estiver pressionado | dispara ação |
| item de menu | destacado enquanto selecionado mas não disparado | dispara ação |
| (grupo de) botões de opção (*radio buttons*) | apenas um fica selecionado de cada vez | define valor a partir de um conjunto de alternativas mutuamente exclusivas |
| (grupo de) botões de seleção (*check boxes*) | zero ou mais selecionados de cada vez | define valor a partir de um conjunto de alternativas inclusivas |
| caixa de lista (*listbox*) | zero ou mais opções selecionadas | define valor(es) dentre alternativas exclusivas ou inclusivas (listas de seleção simples ou múltipla, respectivamente) |
| caixa de texto (*textbox*) | usuário precisa digitar valor | define valor |

*Tabela 4 — Conjunto típico de* widgets *que podem ser utilizados no mapeamento conteúdo–expressão das operações e seus parâmetros.*

Uma associação comum é feita entre itens de menu e a estrutura das operações. Se a estrutura for operação+objeto+complementos, por exemplo, temos um menu para cada operação, um item de menu para cada objeto de cada operação, e um submenu para cada complemento, respectivamente. Ao representar este mapeamento no modelo da aplicação, uma extensão pode ser automaticamente mapeada, mantendo a consistência com o *design* original. Como estamos lidando com a variação de apenas um elemento dos sintagmas, este mapeamento garante a preservação da semântica dos signos de interface resultantes. Caso fosse possível variar mais de um elemento de um sintagma, este significado poderia ser perdido.

A próxima seção descreve algumas situações de uso do nosso protótipo, indicando os cálculos com base metafórica ou metonímica envolvidos em cada passo, juntamente com as classificações relevantes para cada um.

## 4.5 Situações de Uso

Esta seção apresenta algumas situações de uso do protótipo, onde pretendemos ilustrar seu funcionamento através da descrição de algumas situações de uso. São apresentados também os predicados correspondentes aos passos indicados nos *walkthroughs*.

A Tabela 5 mostra os passos para se criar um novo tipo (puppet) no protótipo, a partir de um tipo existente (beeper), tomando emprestados atributos de um outro tipo (lamppost). Os passos indicam também a correspondência de uma linha da tabela com a figura que a sucede.

| PASSO | AÇÃO DO USUÁRIO | RESPOSTA DA INTERFACE |
|---|---|---|
| **1 ( ,** na figura) | seleciona um beeper B | destaca a representação do beeper B |
| **2** | seleciona ferramenta de extensão | alguma indicação de mudança para modo de extensão |
| **3 (❷, na figura)** | | apresenta opções:<br>1. criar novo tipo de objeto com base em B<br>2. criar ou estender operação envolvendo beeper<br>3. criar ou estender relação envolvendo beeper<br>4. criar novo tipo (genérico) |
| **4** | seleciona opção 1 | apresenta caixa de texto para usuário digitar nome do novo tipo, mostra os atributos de B e as opções:<br>1. acrescentar atributo<br>2. remover atributo<br>3. modificar valor de atributo |
| **5 ( , na figura)** | digita “puppet” e seleciona opção 1 | exibe nome do novo tipo, e as opções:<br>1. criar atributo<br>2. pegar atributo de outro tipo |
| **6** | seleciona opção 2 | exibe lista de tipos e a mensagem: “Clique em um elemento ou selecione na lista” |
| **7 (❹, na figura)** | clica em um “lamppost” | exibe lista dos atributos de “lamppost” e a mensagem “Selecione atributo” |
| **8 (❺, na figura)** | seleciona “orientation” | |

| | | |
|---|---|---|
| **9** | clicks on lamppost | exibe opções (idem ao passo 4)<br>1. acrescentar atributos<br>2. remover atributos<br>3. modificar valor de atributo |
| **10** | seleciona opção 3 | exibe atributos do novo objeto e a mensagem: “Selecione atributo” |
| **11** | seleciona “shape” | exibe opções:<br>1. acrescentar nova forma<br>2. modificar representação |
| **12** | seleciona opção 1 | apresenta caixa de texto para dar nome à forma e uma lista de possíveis representações primitivas |
| **13** | digita “puppet-shaped” 3 seleciona “triangle” e “circle” | apresenta um editor gráfico especial para escolher a posição relativa das primitivas e o ponto de referência para orientação |
| **14** | posiciona o círculo no topo do triângulo e seleciona o ponto no topo do círculo como ponto de orientação | acrescenta a representação criada ao conjunto de representações, e exibe as opções (idem ao passo 4):<br>1. acrescentar atributos<br>2. remover atributos<br>3. modificar valor de atributo |
| **15** | seleciona “fim” | |

*Tabela 5 — Situação de uso do protótipo num exemplo onde o usuário deseja criar um novo tipo, a partir de um tipo existente.*

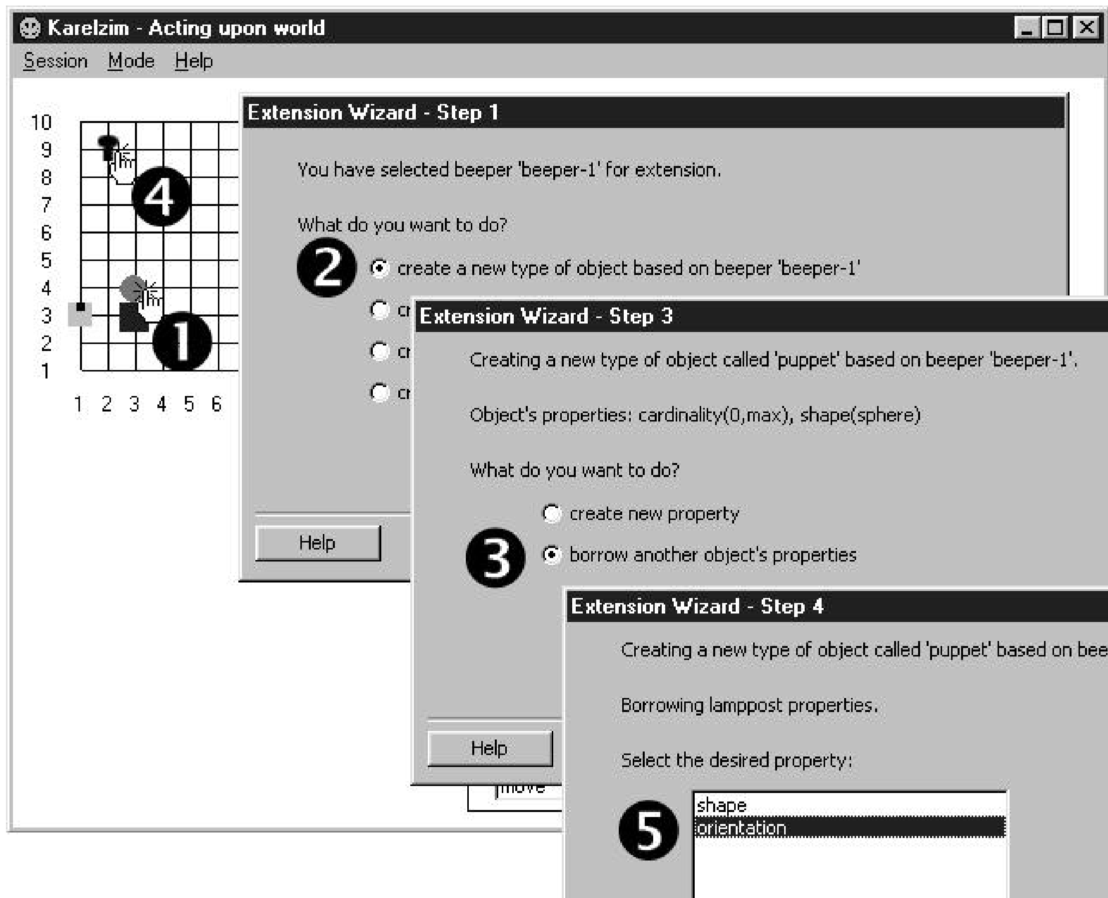


*Figura 8 — Seqüência de telas ilustrando parte da situação de uso da criação de um novo tipo a partir de um tipo existente*

A Tabela 6 mostra os passos para se criar uma operação (get_permit(actor,corner))no protótipo, com base em uma operação existente (borrow(actor, thing)). Os passos indicam também a correspondência de uma linha da tabela com a figura que a sucede.

| PASSO | AÇÃO DO USUÁRIO | RESPOSTA DA INTERFACE |
|---|---|---|
| **1<br>( , na figura)** | Clica em um robô R | destaca representação do robô |
| **2** | Seleciona ferramenta de extensão | alguma indicação de mudança para modo de extensão |
| **3<br>(❷, na figura)** | | apresenta opções:<br>1. criar novo tipo de objeto com base em R<br>2. <u>criar ou estender operação envolvendo robô</u><br>3. criar ou estender relação envolvendo robô<br>4. criar novo tipo (genérico) |

| 4 | Seleciona opção 2 | apresenta caixa de texto para digitar nome da nova operação; exibe mensagem “selecione operação da lista abaixo ou o elemento de interface que a dispara” |
|---|---|---|
| **5 ( , na figura)** | digita “get permit” e clica no item “borrow(Thing, Actor)” | |
| **6 (❹, na figura)** | | exibe o protótipo da operação: borrow(actor, thing) e as opções:<br>1. acrescentar parâmetro<br>2. remover parâmetro<br>3. modificar parâmetro<br>4. restringir execução |
| **7 (❺, na figura)** | seleciona opção 3 | exibe lista de parâmetros (actor e thing) e a mensagem: “Selecione o parâmetro a ser alterado” |
| **8 ( , na figura)** | seleciona “thing” | exibe tipos existentes e a mensagem “Selecione o tipo” |
| **9 ( , na figura)** | seleciona “corner” | exibe nome da nova operação e o protótipo get_permit(actor,corner), e a mensagem: “Esta operação é inválida pois contém um teste envolvendo a relação own(actor, thing), onde “thing” não está definido. O que você quer fazer?<br>1. criar relação own(actor,corner) com base em own(actor,thing)<br>2. considerar “thing” igual a...<br>3. pedir o valor de “thing” em tempo de execução<br>4. considere o teste verdadeiro e execute as instruções a ele subordinadas<br>5. considere o teste falso e ignore as instruções a ele subordinadas |
| **10** | seleciona opção 1 | exibe nome da nova operação e o protótipo get_permit(actor,corner), e a mensagem: “Esta operação é inválida pois contém uma instrução envolvendo a relação may_use(actor, thing), onde “thing” não está definido. O que você quer fazer?<br>1. criar relação may_use(actor,corner) com base na relação may_use(actor,thing)<br>2. considerar “thing” igual a...<br>3. pedir o valor de “thing” em tempo de execução<br>4. ignorar a instrução |

| | | |
|---|---|---|
| **11** | seleciona opção 1 | exibe nome da nova operação e o protótipo get_permit(actor,corner), a mensagem "Esta operação é válida", e as opções:<br>1. acrescentar parâmetro<br>2. remover parâmetro<br>3. modificar parâmetro<br>4. restringir execução<br>5. acrescentar elemento de interface para disparar ação |
| **12** | seleciona opção 5 | exibe opções:<br>1. acrescentar item de menu<br>2. acrescentar botão |
| **13** | seleciona opção 1 | exibe hierarquia de menus aberta em "Ownership", e pede para usuário selecionar posição do novo item neste menu, sugerindo a posição abaixo de Borrow |
| **14** | aceita sugestão | (igual ao passo 9)<br>exibe nome da nova operação e o protótipo get_permit(actor,corner), a mensagem "Esta operação é válida", e as opções:<br>1. acrescentar parâmetro<br>2. remover parâmetro<br>3. modificar parâmetro<br>4. restringir execução<br>5. acrescentar elemento de interface para disparar ação |
| **15** | seleciona "fim" | |

*Tabela 6 — Situação de uso do protótipo num exemplo onde o usuário deseja criar uma nova operação, a partir de uma operação existente.*

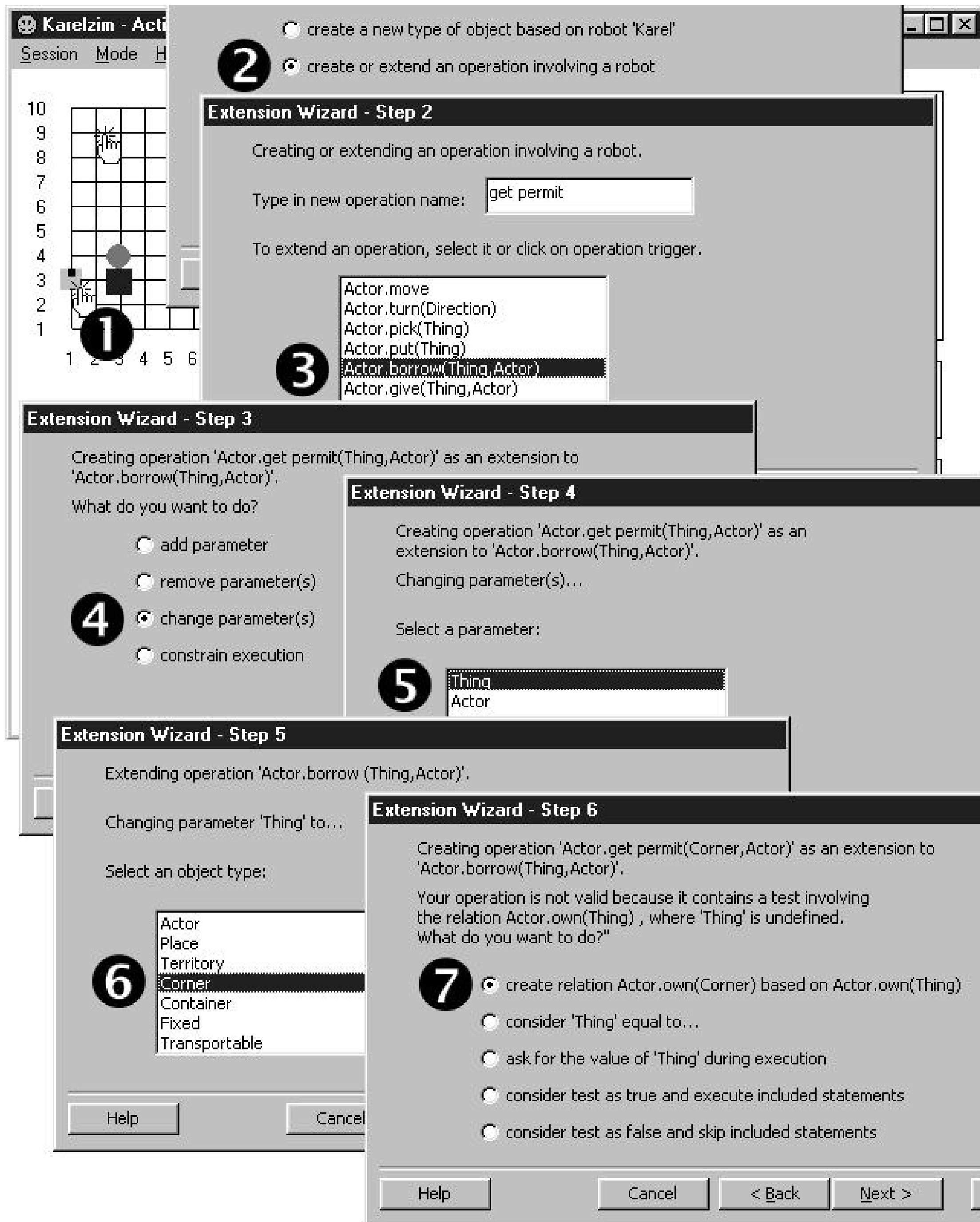


*Figura 9 — Seqüência de telas ilustrando parte da seqüência de uso de criação de uma nova operação a partir de uma operação existente.*

## 4.6 Orientações para o *Designer*

Esta seção apresenta orientações para o *designer* que deseje projetar uma aplicação com base no modelo apresentado neste trabalho. Indicamos classificações de uso freqüente que

podem ser úteis, e esclarecemos algumas implicações de certas decisões, tanto de representação quanto de classificação.

1. Defina o nível básico de categorização.

Crie classificações de acordo com uma análise das distinções relevantes do domínio e para a classe pretendida de tarefas do usuário. Como o significado é determinado por oposições entre os elementos, classificações muito abrangentes não permitem que os mecanismos gerem interpretações ou extensões interessantes, pois considera "tudo igual". Por outro lado, quanto mais específicas forem as classificações, mais elas limitam o escopo dos mecanismos de extensão até, no limite, só haver um tipo de elemento por classificação, onde a única interpretação possível seria a literal. Reproduzimos abaixo o exemplo da página 60.

> **Exemplo**. Sejam os tipos robot, block, toy, e beeper. Consideremos os três grupos de classificação a seguir:
> 1) Stuff = {robot, block, toy, beeper}
> 2) Actors = {robot}, Things = {block, toy, beeper}
> 3) Actors_with_wheels_for_legs = {robot},
> Things_that_are_solid_and_regular_shaped = {block},
> Things_that_are_made_up_of_blocks = {toy},
> Things_that_beep_3_times = {beeper}

2. Defina as operações para os tipos que estiverem mais acima em uma cadeia metonímica. Para além destes, serão geradas iterações com base em quantificadores universais.

Nossos mecanismos de cálculo de metonímia percorrem as possíveis cadeias metonímicas nos dois sentidos. Para baixo (do todo T para uma parte P, por exemplo), substituindo uma ocorrência de T por P, em operações que se apliquem a P e não a T. No sentido oposto (de uma parte P para o todo T, por exemplo), o mecanismo funciona como um gerador de iterações, através de quantificação universal do tipo "para todo P em T..."

**Exemplo**. Seja a operação paint(transportable_object,color) como alterando o valor do atributo color do tipo transportable_object, e os predicados:

is_a(toy, transportable_object)
instance_of(toy1,toy)
part_of(block1,toy1)
part_of(block2,toy1)
part_of(connector1,toy1)
contained(toy1,box1)
attribute_of(color,block)

A operação paint(toy1) é interpretada literalmente. A operação paint(block1) também, visto que color é atributo de block. Já a operação paint(connector1) seria interpretada como paint(toy1), visto que connector1 não possui o atributo color.

Percorrendo a cadeia metonímica no sentido oposto, a operação paint(box1) seria interpretada como "para todo transportable_object em box1, paint(transportable_object,color)", visto que box1 também não possui o atributo color.

3. Como conseqüência de (2), é interessante expressar como relações metonímicas todas as relações que possam dar origem a iterações, como ilustrado abaixo.

Uma relação located poderia ser definida como metonímica, a fim de gerar iterações do tipo "para todo elemento E localizado em L, pick(E)" como interpretação de uma operação pick(L), que não possui significado literal.

Assim, o *designer* deve se dar conta de que o potencial dos cálculos de extensão por metonímia são diretamente proporcionais ao comprimento das cadeias que puderem ser estabelecidas de acordo com as relações metonímicas.

4. Evite criar um conceito unificador.

Como nosso mecanismo procura interpretar as expressões que não possuem sentido literal através da navegação por cadeias metonímicas, se todas estas cadeias atingirem um tipo unificador, este tipo será sempre considerado uma possível interpretação para qualquer tipo de expressão, neutralizando as oposições dentro do domínio. Desta forma, os mecanismos de extensão podem gerar distorções, dando interpretações sem significado para o usuário.

Em nosso protótipo, se criarmos um conceito unificador world, acrescentando os predicados part_of(actor,world), part_of(thing,world) e part_of(place,world), os mecanismos de interpretação poderiar gerar distorções do tipo pick(world,world), que fazem pouco ou nenhum sentido para o usuário e em nada contribuem para suas tarefas.

Caso seja necessário criar um conceito unificador, o mecanismo de cálculo de extensões deverá ser modificado para parar o cálculo por metonímias antes de alcançar este conceito.

5. Explore o mecanismo de classificação para criar sinônimos.

Nosso esquema de classificações pode ser utilizado para criar sinônimos, aumentando a expressividade da linguagem do usuário. Vale observar que, caso dois elementos compartilhem todas as classificações, estes são considerados, pela aplicação, como sinônimos perfeitos, ou co-expressivos, sejam eles tipos, atributos ou valores de atributos.

Sejam {small, medium, large} os valores possíveis do atributo size. A classificação {big, large} permite interpretar expressões da forma size.big como significando size.large.

6. Utilize com cautela a linguagem visual. O nível de articulação dos elementos representados determina o escopo dos operadores, e conseqüentemente limita a expressividade da linguagem.

Numa interface predominantemente visual, há pouca articulação dos signos expressivos, e portanto dificilmente conseguimos formular regras de construção de novos signos (Eco, 1976; Martins, 1998). Em uma aplicação extensível, isto implica a perda de controle das extensões no âmbito expressivo, visto que não temos como restringir ou guiar o usuário por tais extensões. O resultado é uma total liberdade de customização da interface para refletir extensões de funcionalidade, cujo resultado pode ser uma degeneração da interface, como apresentado na Figura 4, na página 40.

7. Utilize os mecanismos de geração de extensões como ferramenta de *design*, para detectar distorções existentes ou oportunidades adicionais de extensão.

Os mecanismos apresentados neste trabalho possuem, além da capacidade interpretativa, uma capacidade gerativa, na medida em que podem gerar o conjunto de expressões

metafóricas e metonímicas e verificar sua validade. É provável que muitas das expressões geradas não sejam plausíveis e, portanto, não tenham grande utilidade. Neste caso, o *designer* pode ajustar as classificações dos elementos no modelo a fim de representar mais precisamente o domínio e as extensões potenciais que ele pretende disponibilizar ao usuário final.

# 5. Conclusões

Este capítulo compara nossa abordagem com trabalhos relacionados, enumera e analisa as contribuições apresentadas para a comunidade científica, descreve suas limitações e lança novos desafios na forma de sugestões para trabalhos futuros.

## 5.1 Discussão

Esta seção apresenta uma avaliação da nossa abordagem com relação a abordagens existentes para aplicações extensíveis, apresentadas na Seção 2.1, "Aplicações extensíveis".

A técnica de gravação de macros consiste em armazenar os passos de interação do usuário, agrupando-os sob um só nome, para reproduzir estes passos em um momento futuro. As seleções do usuário com relação a valores de parâmetros são gravadas como constantes, e as operações em uma seqüência linear de instruções, sem possibilidade de estruturas do tipo condicional ou iteração. Uma extensão gerada utilizando nosso mecanismo de *wizards* não apresenta estas limitações, pois os valores constantes podem

ser substituídos por atributos dos tipos do modelo, e pode-se criar restrições sobre a execução de uma instrução (pré-condições), assim como estruturas de repetição.

Outra técnica comum para programação feita por usuários finais é a configuração de parâmetros. É utilizada geralmente para pequenos conjuntos de configuração sobre o funcionamento de uma aplicação, como por exemplo, se um arquivo deve ser armazenado periodicamente, se um verificador ortográfico em um editor de textos deve verificar cada palavra digitada pelo usuário ou esperar que o usuário dispare explicitamente o verificador, etc. Observe que, neste tipo de extensão, o usuário precisa encontrar o parâmetro que deseja configurar em meio a um grande número de parâmetros.

Uma técnica muito flexível de programação feita por usuários finais é a disponibilização de uma linguagem de programação ou de *script*, geralmente textual, que o usuário pode utilizar para escrever o código da extensão. Entretanto, esta técnica não é adequada a usuários finais leigos em programação, visto que eles não possuem conhecimento sobre conceitos básicos de computação, como variáveis, condicionais e iterações, entre outros (Myers, 1992, *Introduction*). Além disso, grande parte das aplicações extensíveis sequer apresenta o modelo subjacente da aplicação e do domínio ao usuário que, conseqüentemente, não conhece os tipos que pode manipular. Algumas abordagens tentam reduzir este problema, apresentando, a cada passo de interação do usuário, o comando equivalente na linguagem de extensão. Esta técnica de revelação progressiva (ou *progressive disclosure*, segundo [DiGiano e Eisenberg, 1995; DiGiano, 1996]) pretende instruir gradativamente o usuário com relação aos modelos do domínio e às estruturas sintáticas aceitas pela linguagem de extensão. Mas mesmo esta abordagem não oferece orientações sobre os conceitos de programação necessários para a redação manual de código em uma linguagem de programação ou *script*.

Embora nossa abordagem seja mais limitada do que uma linguagem de programação, utilizamos mecanismos inatos aos seres humanos. Comparativamente com linguagens de programação, o que propomos é que, ao invés de se utilizar um conhecimento especializado para uma tarefa especializada —programação—, se utilize uma fonte de conhecimento natural (Lakoff e Johnson, 1980) aplicada a uma tarefa especializada.

As técnicas de programação por exemplos ou por demonstração podem ser classificadas de acordo com a presença ou ausência de mecanismos de inferência. Algumas técnicas de

programação por demonstração fazem inferências sobre um conjunto de interações do usuário com a aplicação, visando generalizar seqüências de passos de interação em uma extensão de funcionalidade (Cypher, 1993b; Kurlander e Feiner, 1993; Lieberman, 1993a). Outras possuem um mecanismo de generalização, deixando a cargo do usuário especificar, em tempo de execução, os valores que tiverem sido abstraídos no mecanismo de generalização e não possuírem valor determinado (Lieberman, 1993b).

Grande parte dos mecanismos de inferência utilizados nas técnicas de programação por demonstração são dependentes do domínio e da aplicação. Nossa abordagem oferece mecanismos genéricos que operam sobre modelos do domínio e da aplicação em que estejam representados os elementos descritos na seção 3.2, "Representação do conhecimento".

Uma das vantagens da nossa abordagem para programação feita por usuários finais é utilizar mecanismos familiares aos usuários, que fazem uso de metáforas para descrever ou interpretar um conceito novo em termos de outro conhecido, ou uso de metonímias para fazer referência a um elemento através de uma relação com outro elemento. Como utilizamos estes tipos de raciocínio ao nos comunicarmos em linguagem natural, torna-se mais fácil transpor estes mecanismos para o ambiente computacional do que inventar novos mecanismos de inferência criados especificamente para este ambiente, em detrimento à compreensão e facilidade de uso pelo usuário em interações com uma aplicação.

Portanto, assumimos que, ao interagir com uma aplicação, o usuário utiliza a linguagem de interface disponível para compor enunciados que fazem sentido para ele, e que ele espera que produzam um determinado resultado, visando a realização de uma tarefa. Sob esta perspectiva comunicativa, tentamos produzir um resultado a partir dos enunciados do usuário que não possuem sentido literal. Para tanto, utilizamos mecanismos que permitem descrever um elemento em termos de outro, de acordo com classificações de semelhança e com alguns tipos de relação. Como estes mecanismos são familiares aos usuários finais e usados em sua comunicação em linguagem natural, muitas vezes sem que eles tenham consciência desta utilização, percebemos que há grandes chances de eles tentarem utilizar estes mesmos mecanismos durante a interação com uma aplicação computacional. Além disto, ao utilizarmos recursos de metáfora e metonímia, obtemos expressões mais

interessantes do que as obtidas com operadores de extensão de natureza puramente lexical ou sintática.

É importante comentar sobre o novo papel do *designer* no projeto das aplicações extensíveis descritas neste trabalho. De criador de uma ferramenta de trabalho, o *designer* passa a ser também criador de uma ferramenta de autoria. O impacto disto é a necessidade de fornecer para os usuários finais ferramentas semelhantes às utilizadas pelo *designer*, mas principalmente suporte à programação a ser feita por não-programadores. Este suporte deve se concentrar na revelação e manipulação dos modelos subjacentes e dos conceitos de programação propriamente ditos. Esta nova perspectiva traz a necessidade de um suporte ainda maior para o próprio *designer*, a fim de ajudá-lo a identificar o potencial de extensão da sua aplicação e fazer os ajustes necessários.

## 5.2 Contribuições

As técnicas existentes para programação feita por usuários finais pouco exploram o potencial comunicativo das linguagens de interação e de extensão presentes nas próprias aplicações. Quando projetadas com objetivos comunicativos em mente, as aplicações permitem uma exploração extensiva de uso de recursos lingüísticos. Em nosso trabalho, exploramos o uso de metáforas e metonímias para fins de extensão da funcionalidade e da interface de aplicações.

Conseguimos trazer para a interface mecanismos de extensão que realizam cálculos de natureza não-lexical sobre os elementos dos modelos de domínio e da aplicação, representados na base de conhecimento. Os cálculos através de operadores metafóricos e metonímicos utilizam alguns dos mecanismos que ocorrem na linguagem natural, o que facilita a compreensão de sua aplicação por parte do usuário.

Dentre as técnicas de programação feita por usuários finais, a programação via interface se situa entre gravação de macros e linguagem de programação. Ela quebra barreiras entre interface e ambiente de extensão, incorporando à linguagem de interface mecanismos meta-lingüísticos de extensão de natureza semântico–pragmática (metáforas e metonímias). A técnica de gravação de macros traz para a interface mecanismos de extensão de natureza puramente lexical. Comparada a linguagens de programação, a programação via interface permite que o usuário se expresse de maneira natural e

contextualizada na tarefa, como proposto por [Nardi, 1993], e procura dar uma interpretação a suas expressões que seja válida no ambiente computacional, dentro do escopo da aplicação.

Além da capacidade interpretativa das expressões do usuário, nossa abordagem oferece ainda um mecanismo de construção de extensões. Ao longo deste processo, conseguimos manter o usuário no controle das mesmas. Nossos mecanismos percorrem os modelos do domínio e da aplicação, calculando possíveis metáforas e metonímias, e sugerindo alternativas. Para facilitar a extensão de aplicações por usuários finais, utilizamos *wizards* para guiá-los neste processo, sugerindo alternativas a partir de uma indicação do próprio usuário sobre a intenção de extensão. O estilo de interação proposto permite uma apresentação incremental dos modelos do domínio e da aplicação, assim como uma gradual introdução de conceitos de programação (DiGiano e Eisenberg, 1995; DiGiano, 1996). Como os mecanismos de extensão são acessados a partir da linguagem de interface, mas operam na linguagem de extensão de forma previsível e controlada, o contínuo semiótico é preservado por definição, pois estes mecanismos garantem que toda extensão feita possua correspondência nestas duas linguagens.

Uma conseqüência do nosso modelo é promover as extensões à categoria de *affordance* (Winograd, 1996; Norman, 1999). As *affordances* de uma aplicação computacional são as facilidades percebidas pelos usuários a partir da interface. A partir do momento em que trazemos o mecanismo de extensões para a interface, ele fica automaticamente ao alcance do usuário, integrado às facilidades intrínsecas da aplicação.

A programação via interface restringe as possíveis extensões, de forma a preservar a consistência entre expressão e conteúdo, tal como especificada pelo *designer* original. Em particular, destacamos a possibilidade de a extensão ser refletida diretamente no *layout* de interface, como descrito na seção “Mapeamento Conteúdo–Expressão de Tipos”. Esta questão ilustra o papel de nossa pesquisa no quadro geral de Engenharia Semiótica. Tal como argumentado em [Barbosa et al., 1999], a programação feita por usuários finais apresenta com clareza os problemas de construção e transmissão de significado em programas computacionais. O ponto central é que, na qualidade de *designers*, precisamos comunicar não apenas a nossa interpretação sobre os conjuntos de tarefas que disponibilizamos através da interface, mas também o contexto desta interpretação.

Somente quando o usuário consegue entender e adotar este contexto, torna-se possível estender consistentemente uma aplicação computacional.

Nosso modelo é genérico o bastante para não limitar o trabalho do *designer*. Ele lhe oferece um ferramental poderoso para embutir semântica em uma aplicação extensível. O *designer* poderá escolher a linguagem de representação e os tipos de classificação mais adequados ao domínio de sua aplicação, desde que satisfaça os requisitos de linguagens de representação e de tipos de classificação definidos pelo modelo. Caberá a ele também regular o nível de extensibilidade oferecido ao usuário, de acordo com a classe-alvo de usuários de sua aplicação.

Tal como levantado por Myers, a programação feita por usuários finais é uma resposta possível para a crise de desenvolvimento de software atual, que procura acumular em uma aplicação todos os seus possíveis cenários de uso (Myers, 1992). Nossa abordagem oferece aos usuários um mecanismo de aquisição de conhecimento limitado mas simples, que permite desenvolver aplicações menores, deixando a cargo do usuário atingir, através de extensões, os cenários de uso específicos a suas necessidades. Uma das conseqüências benéficas desta opção é tornar as aplicações mais fáceis de serem aprendidas, pois apresentam uma interface mais concisa.

## 5.3 Trabalhos Futuros

A pesquisa feita durante o desenvolvimento deste trabalho aponta para alguns desdobramentos de nossa abordagem, como indicaremos nesta seção.

Um dos maiores desafios deste trabalho é sua aplicabilidade em ambientes reais, como agendas, editores de texto e planilhas, entre outros. Já no nosso protótipo, verificamos que, para obtermos extensões interessantes, precisamos de um nível de representação de conhecimento bastante fino. Isto implica que um software comercial deveria passar por um processo de re-engenharia, para incorporar a seus modelos as classificações necessárias para a geração de metáforas e metonímias consistentes com a intenção de *design*.

Uma questão muito importante envolve versões sucessivas de uma aplicação extensível. Uma versão posterior deve permitir que as extensões reais ou potenciais da versão

anterior continuem válidas. Caso contrário, o usuário deverá descartar o modelo aprendido e construir um novo, reiniciando o caminho de aprendizado.

Assim, precisamos fazer mais investigações sobre a escalabilidade do modelo, e sua repercussão sobre o ciclo de vida das aplicações extensíveis.

Na medida em que oferecemos ao usuário maneiras de ele se expressar espontaneamente, conseguimos capturar padrões de usabilidade tais como percebidos pelos usuários. Trabalhos futuros poderiam explorar o uso do nosso modelo para capturar padrões de interação e funcionalidade desejados pelos usuários. Estes padrões podem constituir uma fonte importante para novos princípios de *design*, qualificados pelos destinatários do produto projetado, ou seja, pelos usuários. No cenário de Engenharia Semiótica, consideramos uma aplicação uma comunicação unilateral do *designer* para o usuário. Em aplicações extensíveis, podemos conceber a inversão dos papéis, onde o *designer* passa a ser o receptor de uma mensagem enviada pelo usuário através das extensões formuladas.

Outra questão interessante a ser investigada é a possibilidade de utilizar outros recursos lingüísticos para gerar extensões, através de figuras de linguagem como um todo, e outros mecanismos de natureza pragmática. Este ponto aproxima as linguagens de interface das linguagens "naturais", no sentido de serem aquelas códigos "naturais" de comunicação entre usuários e sistemas.

Um trabalho futuro imediato é a implementação completa deste modelo, visando a coletar dados empíricos que validem ou levem a ajustes as propostas teóricas aqui apresentadas.

# Referências


| | |
|---|---|
| Adler e Winograd, 1992 | Adler, P. e Winograd, T. (eds.) *Usability: Turning Technologies into Tools*. Oxford University Press. New York, NY. 1992. |
| Andersen, 1990 | Andersen, P.B. *A Theory of Computer Semiotics*. Cambridge. Cambridge University Press. 1990. |
| Andersen et al., 1993 | Andersen, P.B.; Holmqvist, B.; Jensen, F.F. (eds.) *The Computer as Medium.* Cambridge University Press. 1993. |
| Apple, 1992 | Apple Computer, Inc. *Macintosh Human Interface Guidelines*. Reading, Ma. Addison Wesley. 1992. |
| Asymetrix, 1994 | Asymetrix, Corporation. *Openscript Reference Manual*. 1994. |

| | |
|---|---|
| Barbosa et al., 1997 | Barbosa, S.D.J.; Cara, M.P.; Cereja,J.R.; Cunha, C.K.V.; de Souza, C.S. “Interactive Aspects in Switching between User Interface Language and End-User Programming Environment: A Case Study”. Em *Proceedings of WOMH’97*. São Carlos, Brazil. 1997 |
| Barbosa et al., 1998 | Barbosa, S.D.J.; da Cunha, C.K.V.; da Silva, S.R.P. “Knowledge and Communication Perspectives in Extensible Applications”. Em *Proceedings of IHC’98*. Maringá, PR. 1998. |
| Barbosa et al., 1999 | Barbosa, S.D.J.; da Silva, S.R.P.; de Souza, C.S. “Extensible Software Applications as a Semiotic Engineering Laboratory”. A ser publicado em *Working Papers in the Semiotic Sciences*. Legas, Ottawa, Canada. 1999. |
| Bergin et al., 1997 | Bergin, J.; Stehlik, M.; Roberts, J.; Pattis, R. *Karel++ A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley and Sons. 1997. |
| Bruce, 1975 | Bruce, B. “Case Systems for Natural Language”. *Artificial Intelligence* 6(4): 327-360. 1975. |
| Brown e Yule, 1983 | Brown, G. e Yule, G. *Discourse Analysis*. Cambridge University Press, 1983. |
| Chang, 1990 | Chang, S.K. *Visual languages and visual programming*. New York. Plenum Press.1990. |
| Chomsky, 1959 | Chomsky, N. “On certain formal properties of Grammars”. *Information and Control* **2**: 2, 137–167. 1959. |
| Cypher, 1993a | Cypher, A. (ed.) *Watch What I Do: Programming by Demonstration*. The MIT Press. Cambridge MA. 1993. |

Cypher, 1993b | Cypher, A. "Eager: Programming Repetitive Tasks by Demonstration". Em Cypher, A. et al. (eds.) *Watch What I Do: Programming by Demonstration*.

da Silva et al., 1997 | da Silva, S.R.P.; Barbosa, S.D.J.; de Souza, C.S. "Communicating Different Perspectives on Extensible Software". Em Lucena, C.J.P. (ed.) *Monografias em Ciência da Computação*. Departamento de Informática. PUC-RioInf MCC 23/97. Rio de Janeiro. 1997.

da Silva e Ierusalimschy, 1997 | da Silva, S.R.P.; de Souza, C.S.; Ierusalimschy, R. "A Communicative Approach to End-User Programming Languages". Em Lucena, C.J.P. (ed.) *Monografias em Ciência da Computação*. Departamento de Informática. PUC-RioInf MCC 47/97. Rio de Janeiro. 1997.

de Souza, 1993 | de Souza, C.S. "The Semiotic Engineering of User Interface Languages". *International Journal of Man-Machine Studies*. No. 39. pp. 753-773. 1993.

de Souza, 1996 | de Souza, C.S. "The Semiotic Engineering of Concreteness and Abstractness: from User Interface Languages to End-User Programming Languages". Em Andersen, P.; Nadin, M.; Nake, F. (1996) *Informatics and Semiotics*. Dagstuhl Seminar Report No. 135, p. 11. Schloß Dagstuhl., Germany.

de Souza, 1997 | de Souza, C.S. "Supporting End-User Programming with Explanatory Discourse". Em *Proceedings of Intelligent Systems and Semiotics*, Maryland, USA. September 22-25, 1997. pp. 461-466. 1997.

de Souza, 1999 | de Souza, C.S. "Semiotic engineering principles for evaluating end-user programming environments". Em

| | |
|---|---|
| | Lucena, C.J.P. (ed.) Monografias em Ciência da Computação. Departamento de Informática. . PUC-RioInf MCC 10/99. Rio de Janeiro. 1999. |
| de Souza e Barbosa, 1996 | de Souza, C.S. e Barbosa, S.D.J. "End-User Programming Environments: The Semiotic Challenges". Em Lucena, C.J.P. (ed.) Monografias em Ciência da Computação. Departamento de Informática. . PUC-RioInf MCC 19/96. Rio de Janeiro. 1996. |
| DiGiano, 1996 | DiGiano, C. " A vision of highly-learnable end-user programming languages". *Child's Play '96 Position Paper*. 1996. |
| DiGiano e Eisenberg, 1995 | DiGiano, C. e Eisenberg, M. "Self-disclosing design tools: A gentle introduction to end-user programming". Em *Proceedings of DIS '95*. Ann Arbor, Michigan. August 23-25, 1995. ACM Press. 1995. |
| Draper, 1986 | Draper, S. "Display Managers as the Basis for User-Machine Comunication". Em D. Norman e S. Draper (eds.) *User Centered System Design*. Hillsdale, NJ. Lawrence Erlbaum. 1986. Pp.339-352. |
| Eco, 1979 | Eco, U. "The Semantics of Metaphor". Em Eco, U. *The Role of the Reader*. Indiana University Press. Bloomington IN. pp.67-89. 1979. |
| Eco, 1986 | Eco, U. *Semiotics and the Philosophy of Language*. Indiana University Press. Bloomington IN. 1986. |
| Eco, 1976 | Eco,U. *A Theory of Semiotics*. Bloomington. Indiana University Press. 1976. |

Eisenberg, 1995 — Eisenberg, M. “Programmable Applications: Interpreter Meets Interface”. *SIGCHI Bulletin*. Apr. Vol. 27(2), ACM Press. 1995.

Fillmore, 1968 — Fillmore, C. “The case for case”. Em *Universals in Linguist Theory*, ed. E. Bach e R.T. Harms. New York, Holt. 1968.

Fischer, 1998 — Fischer, G. “Beyond ‘Couch Potatoes’: From Consumers to Designers”. In *Proceedings of the 5th Asia Pacific Computer–Human Interaction Conference*. IEEE Computer Society. 1999. pp.2–9.

French, 1995 — French, R. *The Subtlety of Sameness*. The MIT Press. Cambridge MA. 1995.

Furtado, 1992 — Furtado, Antonio L. “Analogy by Generalization – and the Quest of the Grail”. *ACM SIGPLAN Notices*, Volume 27, No. 1, January 1992.

Gelernter e Jagannathan, 1990 — Gelernter, D. e Jagannathan, S. *Programming Linguistics*. Cambridge, Ma. MIT Press. 1990.

Gibbs, 1993 — Gibbs, R. “Making Sense of Tropes”. Em Ortony (ed.) *Metaphor and Thought*, 2nd Edition. Cambridge University Press. 1993.

Ghezzi e Jazayeri, 1987 — Ghezzi,C.; Jazayeri, M. *Programming Language Concepts, 2nd Edition*. John Wiley and Sons, Inc. 1987.

Halasz e Moran, 1982 — Halasz, F. e Moran, T.P. “Analogy Considered Harmful”. Em *Human factors in computer systems, conference proceedings*. Gaithersubrg, Maryland. ACM Press. 1982.

| | |
|---|---|
| Hintikka, 1997 | Hintikka,J. *What is Abduction? The fundamental problem of contemporary epistemology.* Manuscrito não publicado. |
| Hjelmslev, 1963 | Hjelmslev, L. *Prolegomena to a Theory of Language*. Menascha: Wincosin University Press, 1963. |
| Hofstadter et al., 1995 | Hofstadter, D. et al. *Fluid Concepts and Creative Analogies*. Basic Books, A Division of HarperCollins Publishers, Inc. New York NY. 1995. |
| Hofstadter, 1979 | Hofstadter, D.R. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books. New York NY. 1979. |
| Holyoak e Thagard, 1996 | Holyoak, K.J. e Thagard, P. *Mental Leaps: Analogy in Creative Thought*. Cambridge, MA. The MIT Press. 1996. |
| Hopcroft e Ullman, 1979 | Hopcroft, J. e Ullman, J. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley. Reading, MA. 1979. |
| Kurlander e Feiner, 1993 | Kurlander, D. e Feiner, S. "A History-Based Macro by Example System". Em *Watch What I Do: Programming by Demonstration* edited by Allen Cypher.MIT Press. Cambridge, Ma. 1993. |
| Lakoff, 1987 | Lakoff, G. *Women, Fire, and Dangerous Things.* The University of Chicago Press. Chicago. 1987. |
| Lakoff, 1993 | Lakoff, G. "The contemporary theory of metaphor". Em Ortony, A. (ed.) *Metaphor and Thought, 2nd edition*. Cambridge University Press. Cambridge MA, pp. 202-251. 1993. |
| Lakoff e Johnson, 1980 | Lakoff, G. e Johnson, M. *Metaphors We Live By.* The University of Chicago Press. Chicago. 1980. |

| | |
|---|---|
| Leite, J.C., 1998 | Leite, J.C. *Modelos e Formalismos para a Engenharia Semiótica de Interfaces de Usuário*. Tese de Doutorado. Departamento de Informática, PUC-Rio. Outubro de 1998. |
| Lieberman, 1993a | Lieberman, H. "Mondrian: A Teachable Graphical Editor". Em Cypher, A. et al. (eds.) *Watch What I Do: Programming by Demonstration*. |
| Lieberman, 1993b | Lieberman, H. "Tinker: A Programming by Demonstration System for Beginning Programmers" . Em Cypher, A. et al. (eds.) *Watch What I Do: Programming by Demonstration*. |
| Lyons, 1981 | Lyons, J. *Language and Linguistics*. Cambridge University Press, 1981. |
| Malone, 1995 | Malone, T.W.; Lai, K. and Fry, C. "Experiments with Oval: A Radically Tailorable Tool for Cooperative Work". In *ACM Transactions on Information Systems*. Vol. 13, No. 2, April 1995. pp.177–205. |
| Martins, 1998 | Martins, I.H. *Um Instrumento de Análise Semiótica para Linguagens Visuais de Interfaces*. Tese de Doutorado. Computer Science Department, PUC-Rio, Brazil. 1998. |
| Microsoft, 1994 | Microsoft Corporation. *Microsoft® Word Developer's Kit, Second Edition*. Redmond. Microsoft Press. 1994. |
| Microsoft, 1995a | Microsoft Corporation. *Microsoft Visual Basic Language Reference*. 1995. |
| Microsoft, 1995b | Microsoft Corporation. *The Windows Interface Guidelines for Software Design*. Redmond. Microsoft Press. 1995. |

| | |
|---|---|
| Myers, 1992 | Myers, B.A. *Languages for Developing User Interfaces*. London. Jones and Bartlett Publishers, Inc. Boston. 1992. |
| Myers et al., 1992 | Myers, B.; Canfield Smith, D.; e Horn, B. "Report of the End User Programming Working Group". Em Myers, B. (Ed.) Languages for Developing User Interfaces. Boston. Jones and Bartlett. pp. 343-366. 1992. |
| Nadin, 1988 | Nadin, M. "Interface Design and Evaluation – Semiotic Implications". Em Hartson, R. e Hix, D. (eds.), *Advances in Human-Computer Interaction*, Volume 2, 45-100. 1988. |
| Nardi, 1993 | Nardi, B. *A Small Matter of Programming*. The MIT Press. Cambridge MA. 1993. |
| Nielsen, 1993 | Nielsen, J. *Usability Engineering*. Academic Press, 1993. |
| Norman, 1986 | Norman, D.A. Cognitive Engineering. Em Norman, D.A. e Draper, S. (eds.) *User-Centered Systems Design*. Lawrence Erlbaum and Associates. Hillsdale, NJ. pp.31-61. 1986. |
| Norman, 1999 | Norman, D.A. " Affordances and Design". Publicação eletrônica, em http://www.jnd.org/dn.mss/ affordances-and-design.html. |
| Nöth, 1997 | Nöth, W. "Representation in semiotics and in Computer Science". Em *Semiotica*, 115-3/4, 203-213. |
| Oliveira, 1999 | Oliveira, D.A.S. "Uma Abordagem Semiótica para a Aquisição e Representação de Conhecimento". Tese de Doutorado a ser defendida em Maio de 1999. |

Ortony, 1993 — Ortony, A. (ed.) *Metaphor and Thought*, 2nd Edition. Cambridge University Press. 1993.

Pattis, 1995 — Pattis,R.E.; Roberts,J.; e Stehlik,M. *Karel the Robot: A Gentle Introduction to the Art of Programming*. New York, N.Y. John Wiley and Sons. 1995.

Peirce, 1931 — Peirce, C.S. *Collected Papers*. Cambridge, Ma. Harvard University Press. (extraído de Buchler, Justus, ed., Philosophical Writings of Peirce, New York: Dover, 1955) 1931.

Preece et al., 1994 — Preece, J.; Rogers, Y.; Sharp, E.; Benyon, D.; Holland, S.; Carey, T. *Human-Computer Interaction*. Addison-Wesley, 1994.

Pressman, 1996 — Pressman, R. *Software Engineering. A Practitioner's Approach*, Fourth edition. MacGraw Hill, 1996.

Rich e Knight, 1994 — Rich, E e Knight, K. *Inteligência Artificial.* 2ª Edição. Makron Books do Brasil. 1994 (tradução brasileira).

Schmalhofer et al., 1995 — Schmalhofer, F.; Kühn, O. e Boschert, S. "The Acquisition and Utilization of Knowledge in Beginners and Advanced Learners". Em Wender, Schmalhofer e Böcker (eds.) *Cognition and Computer Programming*. Ablex Publishing Corporation. Norwood, NJ. 1995.

Saussure, 1916 — Saussure, F. de. *Cours de Linguistique Générale*. Paris, Payot. 1916.

Schank, 1973 — Schank, R.C. "Identification of Conceptualizations Underlying Natural Language". Em Schank & Colby (eds.) *Computer Models of Thought and Language*. Freeman, San Francisco. 1973.

| | |
|---|---|
| Schank, 1975 | Schank, R.C. *Conceptual Information Processing*. North-Holland, Amsterdam. 1975. |
| Suchman, 1987 | Suchman, L. *Plans and Situated Actions: The Problems of Human-Machine Communication*. Cambridge, MA. Cambridge University Press. 1987. |
| Thagard e Verbeurgt, 1998 | Thagard, P. e Verbeurgt, K. “Coherence as Constraint Satisfaction”. *Cognitive Science Journal. 1998.* (Resumo disponível em http://www.umich.edu/ ~cogsci/abstract/5-98thagard.html) |
| Weber e Bögelsack, 1995 | Weber, G. e Bögelsack, A. “Representation of Programming Episodes in the ELM Model”. Em Wender, Schmalhofer e Böcker (eds.) *Cognition and Computer Programming*. Ablex Publishing Corporation. Norwood, NJ. 1995. |
| Winograd, 1996 | Winograd, T. (Ed.) *Bringing Design to Software.* Addison-Wesley, 1996. |
| Witschital, 1995 | Witschital, P. “TRAPS – An Intelligent Tutoring Environment ofr Novice Programmers”. Em Wender, Schmalhofer e Böcker (eds.) *Cognition and Computer Programming*. Ablex Publishing Corporation. Norwood, NJ. 1995. |
| Zloof, 1981 | Zloof, M. QBE/OBE: A language for office and business automation. *IEEE Computer*. May-1981. pp. 13-22. 1981. |

# Apêndice A – Glossário

Este apêndice apresenta alguns dos termos utilizados nesta tese cujas definições são importantes para o entendimento da mesma, e que por esse motivo desejamos clarificar o máximo possível.

## A

| | |
|---|---|
| abdução | tipo de raciocínio que, a partir de um resultado, consiste em gerar um caso e uma hipótese explicativa plausível sobre como chegar ao resultado a partir do caso gerado |
| *affordance* | facilidade percebida pelo usuário a partir da interface (Winograd, 1996; Norman, 1999) |
| ambiente | parte da aplicação que possui interface completa, como menus, diálogos e uma identidade visual própria, e um estado (e.g. ambiente de interação vs. ambiente de extensão) |
| analogia | tipo de raciocínio associativo, com base em relações e semelhanças entre conceitos. |

| | |
|---|---|
| | Utilizamos este termo aqui para descrever o tipo de raciocínio que envolve o uso de metáforas ou metonímias. |
| aplicação estendida | aplicação original* mais o conjunto de operações acrescentadas pelo usuário final e disponibilizadas através de extensões na interface |
| aplicação extensível | aplicação que pode ser modificada pelo usuário, seja por configuração de parâmetros, por gravação de macro, por demonstração, por programação ou qualquer outro mecanismo |
| aplicação original | aplicação que ainda não foi estendida pelo usuário, ou ainda o conjunto de operações primitivas disponíveis para o usuário através de interação com a aplicação |

C

| | |
|---|---|
| configuração, programação por | quando a aplicação oferece diversas opções que o usuário pode selecionar para configurar a aplicação; não é uma tarefa de programação propriamente dita, mas se encaixa no conceito de *end-user programming* |
| contínuo semiótico entre linguagens | duas linguagens são semioticamente contínuas se uma delas (L1) puder ser considerada linguagem de descrição semântica da outra (L2) e se, além disso, a linguagem de descrição semântica (L1) contiver um componente pragmático que possa distinguir quais os *tokens* da linguagem descrita (L2) que fazem sentido |

D

| | |
|---|---|
| dedução | tipo de raciocínio que, partindo de um caso e uma regra, alcança um resultado |
| *design* | atividade de projetar uma aplicação e/ou sua interface |
| *designer* | profissional envolvido no desenvolvimento de software ou de interface de software, em algum nível conceitual ou de especificação |
| domínio | área de conhecimento que envolve todas as tarefas de um determinado usuário |

E

| | |
|---|---|
| *end-user programming* | programação feita por usuários finais |
| engenharia semiótica | área que estuda os processos comunicativos envolvidos em interfaces gráficas e na interação humano–computador |
| extensão | programação feita por usuários finais |

F

| | |
|---|---|
| funcionalidade | tudo o que a aplicação é capaz de fazer; conjunto de operações disponíveis para o usuário ou internas à aplicação |

I

| | |
|---|---|
| ícone | signo cuja relação expressão–conteúdo é de semelhança |
| índice | signo cuja relação expressão–conteúdo é de causalidade |
| indução | tipo de raciocínio que, partindo de um caso e um resultado, procura inferir uma regra |
| interpretante | o conceito que se forma na mente do interpretador de um signo, e que por sua vez pode se tornar um signo em um novo processo de interpretação, causando semiose ilimitada |

M

| | |
|---|---|
| metáfora | recurso lingüístico no qual a significação literal de uma palavra é substituída por outra, em virtude de uma relação de semelhança |
| metonímia | recurso lingüístico que consiste em designar um objeto por uma palavra designativa de outro objeto, que tem com o primeiro uma relação de causa e efeito, de continente e conteúdo, de lugar e produto, de matéria e objeto, de autor e obra, etc. |
| modelo | representação da estrutura e comportamento de um sistema, com a qual se procura explicar ou prever, dentro de um quadro teórico, as propriedades do sistema |
| modelo da aplicação | modelo que define a funcionalidade da aplicação |

## O

| | |
|---|---|
| operação | qualquer ação realizada pelo sistema, ativada através de um elemento de interface ou através da linguagem de extensão |

## P

| | |
|---|---|
| programação feita por usuários finais | qualquer processo de customização ou extensão feito por usuários finais |
| *programming by demonstration* | programação na qual, a partir de uma gravação de macro, a aplicação generaliza os comandos gravados para gerar um programa. Pode utilizar ou não mecanismos de inferência, e diferentes fontes de conhecimento |

## R

| | |
|---|---|
| *representamen* | expressão do signo |

## S

| | |
|---|---|
| semântica | estudo dos significados das palavras e expressões |
| semiose | ato de interpretação de um signo, ou seja de geração de um interpretante a partir da expressão do signo ou *representamen** |
| semiótica | estudo dos signos e processos de significação e de comunicação |
| símbolo | signo cuja relação expressão–conteúdo é convencionada |
| signo | "algo que quer dizer alguma coisa para alguém"; pode ser ícone*, índice* ou símbolo* |
| sintaxe | estudo da estrutura e disposição de palavras em uma sentença |
| sistema | aplicação |

## T

| | |
|---|---|
| tipo | abstração de um conjunto de instâncias que compartilham atributos e relações com outras instâncias e tipos |

| | |
|---|---|
| *token* | uma instância de um tipo |
| *type* | tipo |

U

| | |
|---|---|
| usabilidade | medida associada à facilidade de aprendizado, eficiência de uso, capacidade de retenção do que foi aprendido, baixo índice de erros e satisfação dos usuários (Nielsen, 1993) |
| usuário final | usuário que interage com a aplicação, possivelmente leigo em conceitos de programação |

W

| | |
|---|---|
| *wizard* | seqüência de diálogos que guia o usuário visando a realização de uma determinada tarefa ou operação complexa |

# Apêndice B – Listagens

Este apêndice apresenta algumas listagens em Prolog do nosso protótipo, a fim de ilustrar como os mecanismos de extensão apresentados ao longo deste trabalho podem ser implementados nesta linguagem.

### Representação dos Elementos do Modelo Estático

As listagens nesta seção ilustram a representação dos elementos do modelo estático:

```
dc_type(robot).
dc_type(beeper).
dc_type(transportable_object).
dc_type(nil).
dc_type(thing).

dc_is_a(toy,transportable_object).
dc_is_a(beeper,transportable_object).
dc_is_a(box,transportable_object).
dc_is_a(box,container).
dc_is_a(corner,container).
dc_is_a(bin,container).
dc_is_a(bag,container).
dc_is_a(street,road).
dc_is_a(avenue,road).
dc_is_a(bin,fixed_object).
```

```
dc_is_a(wall,fixed_object).
dc_is_a(lamppost,fixed_object).
dc_is_a(fixed_object,thing).
dc_is_a(robot,actor).
dc_is_a(territory,place).
dc_is_a(container,place).
dc_is_a(transportable_object,thing).

is_a(X,Y) :- dc_is_a(X,Y).

subtype(SubTipo,Tipo) :- is_a(SubTipo,Tipo).
subtype(SubTipo,Tipo) :- is_a(SubTipo,X), subtype(X,Tipo).

dc_part_of(block,toy).
dc_part_of(connector,toy).
dc_part_of(road,territory).
dc_part_of(corner,street).
dc_part_of(corner,avenue).
dc_part_of(bag,robot).

part_of(X,Y) :- dc_part_of(X,Y).

part_of(X,X) :- !,fail.
part_of(SubTipo,Tipo) :- part_of(SubTipo,X),part_of(X,Tipo).

dc_relation(contained,transportable_object,container).
dc_relation(contained,thing,corner).
```

*Listagem 1 — Representação da hierarquia de tipos do protótipo, incluindo herança, composição e relações metonímicas do tipo conteúdo–continente.*

```
dc_attribute(size).
dc_attribute(length).
dc_attribute(shape).
dc_attribute(color).

dc_value_of(small,size).
dc_value_of(medium,size).
dc_value_of(large,size).

dc_value_of(long,length).
dc_value_of(short,length).

dc_value_of(sphere,shape).
dc_value_of(cube,shape).

dc_value_of(red,color).
dc_value_of(green,color).
dc_value_of(blue,color).

dc_classification(appearance,[shape,color]).
dc_classification(measure,[size,length]).

dc_classification(big_things,[big,large,long]).
```

*Listagem 2 — Representação de atributos e suas classificações.*

```
instance_of(beeper1,beeper).
```

```
instance_of(karel,robot).
instance_of(connector1,connector).
instance_of(nil,nil).
instance_of(box1,box).
```

*Listagem 3 — Algumas instâncias definidas na base.*

### Representação dos Elementos do Modelo Dinâmico

A listagem abaixo ilustra a representação dos elementos do modelo dinâmico:

```
dc_operation(move,actor,nil,nil).
dc_operation(pick, actor, transportable_object, nil, nil).
dc_operation(put,actor,transportable_object,nil,nil).
dc_operation(face, actor, thing, nil, nil).
dc_operation(borrow, actor, thing,nil,nil).
dc_operation(lend, actor, thing,nil,nil).
dc_operation(return, actor, thing,nil,nil).

dc_relation(is_with,thing,actor).
dc_relation(belongs_to,thing,actor).
dc_relation(built_by,thing,actor).
dc_relation(lives,actor,place).

dc_classification(ownership,[is_with,belongs_to]).

dc_op_classification(result, [move,face,turn], [ptrans])
dc_op_classification(result, [pick, put,borrow,lend,return], [atrans])
dc_op_classification(result, [build], [build])
```

*Listagem 4 — Representação de relações, operações e algumas classificações.*

### Cálculo de Metonímias

Esta seção apresenta uma listagem que ilustra o cálculo de metonímias possíveis entre dois tipos do modelo.

```
possible_metonymy(A,B,[A,B]) :- m_rel(A,B).
possible_metonymy(A,B,[A|L]) :- mchain(A,X,L1), m_rel(X,B),append(L1,[B],L).
possible_metonymy(A,B,[A|L]) :- m_rel(A,X), mchain(X,B,L1),append([X],L1,L).
possible_metonymy(A,B,[A|L]) :- mchain(A,TA,L1), m_rel(TA,TTA),
        mchain(TTA,B,L2),append(L1,[TTA|L2],L).

mchain(A,B,[X|L]) :- m_rel(A,X), mchain(X,B,L).
mchain(A,B,[X|L]) :- is_a(A,X), mchain(X,B,L).
mchain(A,B,[B]) :- m_rel(A,B).
mchain(A,B,[B]) :- is_a(A,B).


m_rel(A,B) :- part_of(A,B).
m_rel(A,B) :- contained(A,B).
```

*Listagem 5 — Cálculo de Metonímias.*

## Cálculo de Metáforas

Esta seção apresenta uma listagem que ilustra o cálculo de metáforas possíveis entre elementos do modelo, com base em sua classificação.

```
same_classification(X,Y,C) :-
        classification(C,Set1),
        member(X,Set1),
        member(Y,Set1),
        X \== Y.


same_classification_among(Set,X,Y,C) :-
        classification(C,Set1),
        member(X,Set1),
        member(Y,Set1),
        member(Y,Set),
        X \== Y.

all_same_classification_among(Set,X,List,C) :-
        findall(Y,same_classification_among(Set,X,Y,C),List).
```

*Listagem 6 — Cálculo de Metáforas com base nas classificações dos elementos da base.*

## Análise de Instruções

Esta seção apresenta uma listagem do mecanismo de análise de instruções, visando dar uma interpretação a expressões que não possuem interpretação literal válida. Este mecanismo considera instruções contendo atribuições, condicionais e chamadas de operações.

```
analyze_instruction([]).
analyze_instruction([if|T]) :-
        chop(T,NewT,'then'),
        analyze_boolean_expression(NewT).
analyze_instruction([Operation,'(',A,',',O,',',Ia,'.',Iv,',',F|T]) :-
        analyze_expression([Ia,'.',Iv]).
        analyze_operation(Operation, A, O, I, F).
analyze_instruction([Operation,'(',A,',',O,',',I,',',F|T]) :-
        analyze_operation(Operation, A, O, I, F).

analyze_instruction([else]).
analyze_instruction([fi]).
analyze_instruction([RValue,':='|T]) :-
        analyze_rvalue(RValue),!,
        analyze_expression(T).
```

```
analyze_boolean_expression(Exp) :-
        boolean_operator(Operator),
        append(R,[Operator|L],Exp),
        analyze_expression(R),!,
        analyze_expression(L),!.

analyze_rvalue(_).

boolean_operator(X) :-
        member(X,['==','<','>','<=','>=','<>']).
operator(X) :-
        member(X,['+','-','*','/']).

analyze_expression(Exp) :-
        operator(Operator),
        append(R,[Operator|L],Exp),
        analyze_expression(R),!,
        analyze_expression(L),!.
analyze_expression(Exp) :-
        analyze_token(Exp),!,
        analyze_token2a(Exp),!,
        analyze_token3(Exp),!.

analyze_token2a([X,'.',Y]):-
        debugwrite(0,"token %w.%w\n",[X,Y]),
        of(Y,X,T).

/* extensão por metáfora: 2a */
analyze_token2a([X,'.',Y]):-
        debugwrite(0,"token %w.%w\n",[X,Y]),
        findall(VV, of(VV,X,_), LV),
        of(P,X,T),
        classification(C,LC), member(YVal,LC), member(YVal,LV),
        swritef(S,"change '%w' to %w, among valid %w of %w: %w\n",[Y,YVal,T,X,LV]),
        add_user_option(S,true),
        analyze_token2([X,'.',Y]).
analyze_token2a(_).

analyze_token2([X,'.',Y]):-
        debugwrite(0,"token %w.%w\n",[X,Y]),
        findall(VV, of(VV,X,_), LV),
        of(P,X,T),
        swritef(S,"change '%w' to another valid %w of %w, one of %w\n",[Y,T,X,LV]),
        add_user_option(S,true),
        analyze_token2b([X,'.',Y]).
analyze_token2(_).

analyze_token2b([X,'.',Y]):-
        swritef(S2,"%w.%w",[X,Y]),
        analyze_token([S2]).
analyze_token2b(_).

analyze_token3([X,'.',Y,'.',Z]):-
        debugwrite(1,"token %w.%w.%w\n",[X,Y,Z]),
        of(Z,Y,X).
analyze_token3([X,'.',Y,'.',Z]) :-
        analyze_token3a([X,'.',Y,'.',Z]),
        analyze_token3b([X,'.',Y,'.',Z]),
        analyze_token3c([X,'.',Y,'.',Z]),
        analyze_token3d([X,'.',Y,'.',Z]),
```

```
        analyze_token3z([X,'.',Y,'.',Z]).
analyze_token3(_).

/* extensões por metáfora: 3a e 3b */
analyze_token3a([X,'.',Y,'.',Z]):-
        debugwrite(0,"a) token %w.%w.%w\n",[X,Y,Z]),
        of(Y,X,attribute),
        findall(VV, of(VV,Y,X), LV),
        findall([Value,Classification],
            same_classification_among(LV,Z,Value,Classification),LO),
        nonempty(LO),
        swritef(S,"change '%w' to another valid value of attribute '%w', in classification
        %w\n",[Z,Y,LO]),
        add_user_option(S,true).
analyze_token3a(_).

analyze_token3b([X,'.',Y,'.',Z]):-
        debugwrite(0,"b) token %w.%w.%w\n",[X,Y,Z]),
        of(Y,X,attribute),
        findall(VV, of(VV,Y,X), LV),
        same_classification_among(LV,X,Y,C),
        nonempty(LV),
        swritef(S,"change '%w' to another valid value of attribute '%w' of type '%w', in
        classification %w\n",[Z,Y,X,LV]),
        add_user_option(S,true).
analyze_token3b(_).

analyze_token3c([X,'.',Y,'.',Z]):-
        debugwrite(0,"c) token %w.%w.%w\n",[X,Y,Z]),
        not of(Y,X,attribute),
        findall(AA ,of(AA,X,attribute), LA),
        nonempty(LA),
        swritef(S,"change '%w' to another valid attribute of type '%w', one of %w\n",[Y,X,LA]),
        add_user_option(S,true).
analyze_token3c(_).

analyze_token3d([X,'.',Y,'.',Z]):-
        debugwrite(0,"d) token %w.%w.%w\n",[X,Y,Z]),
        of(Y,X,attribute),
        findall(VV, of(VV,Y,X), LV),
        swritef(S2,"add value '%w' to set of valid values %w of attribute '%w' of type
        '%w'\n",[Z,LV,Y,X]),
        add_user_option(S2,true).
analyze_token3d(_).

analyze_token3z([X,'.',Y,'.',Z]):-
        swritef(S2,"%w.%w.%w",[X,Y,Z]),
        analyze_token([S2]).
analyze_token3z(_).

analyze_token([Token]) :-
        debugwrite(1,"analyzing token %w\n",[Token]),
        def(Token),!.

/* substituição puramente sintática */
analyze_token([Token]) :-
        debugwrite(0,"analyze token %w\n",[Token]),
        debugwrite(0,'undefined: %w\n',[Token]),
        analyze_token_classification(Token),
        swritef(S,"ask user for value of %w at run-time",[Token]),
```

```
        add_user_option(S,true),
        swritef(S2,"consider %w constant and equal to...",[Token]),
        add_user_option(S2,true).
analyze_token(_).

analyze_token_classification(X) :-
        debugwrite(0,"analyzing classifications of %w\n",[X]),
        findall(Defined,def(Defined),ListDefined),
        findall([Y,C],same_classification_among(ListDefined,X,Y,C),LV),
        nonempty(LV),
        swritef(S,"change '%w' to value in classification [v,c] one of %w\n",[X,LV]),
        add_user_option(S,true).
analyze_token_classification(_).
```

*Listagem 7 — Cálculo de extensões possíveis, levando em conta relações metonímicas, classificações que dão origens a metáforas, e interpretação puramente sintática.*