Sérgio Roberto Pereira da Silva

UM MODELO SEMIÓTICO PARA PROGRAMAÇÃO POR USUÁRIOS FINAIS

TESE DE DOUTORADO

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 30 de março de 2001.

Sergio Roberto Pereira da Silva

UM MODELO SEMIÓTICO PARA PROGRAMAÇÃO POR USUÁRIOS FINAIS

Tese apresentada ao Departamento de Informática da PUC/RJ como parte dos requisitos para a obtenção do título de Doutor em Informática: Ciência da Computação

Orientadora Clarisse Sieckenius de Souza

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 30 de março de 2001.

Para meus pais, João e Valcira e minha esposa Alessandra, com muito carinho.

AGRADECIMENTOS

À minha Orientadora, Professora Clarisse Sieckenius de Souza, por ter confiado em minha capacidade de realizar este projeto, pela orientação e incentivo e pelas broncas na hora certa, mas, principalmente, pela pessoa humana que é e por ter me mostrado que toda idéia, por mais ingênua que seja, merece atenção.

Aos colegas do Departamento de Informática da PUC-Rio, em especial aos membros do *Semiotic Engineering Research Group* (*SERG*) pelos bons momentos de discussão, que muito contribuíram para o resultado deste projeto, e também pelos momentos de diversão juntos, que ajudaram a superar os momentos difíceis que sugiram. Em especial, à Simone pela leitura crítica do original e pela discussão das idéias centrais deste projeto.

Aos amigos do Laboratório de Métodos Formais, que nestes anos todos compartilham comigo o mesmo espaço de trabalho, em especial ao Professor Armando Haeberer por ter possibilitado o uso deste espaço, após meu desligamento do laboratório.

Ao pessoal do Departamento de Informática da Universidade Estadual de Maringá pelo esforço extra realizado para cobrir minha ausência. Em especial à professora Itana, por ter me incentivado a realizar este projeto, a professora Elisa, por ter agüentado minhas lamentações na sua fase final, e ao professor Ronaldo por me substituir em uma disciplina proporcionando-me mais tempo livre para concluir este projeto.

À CAPES, pela ajuda financeira recebida durante a execução deste projeto.

A toda minha família, em especial a meus pais João e Valcira por terem me proporcionado os três elementos principias na vida de uma pessoa, Amor, Saúde e Educação, e por terem sempre me incentivado no caminho do conhecimento.

À minha esposa Alessandra, pela indispensável ajuda com o texto deste trabalho, e, principalmente, por todo amor, companheirismo, carinho, incentivo e igualmente pela paciência para superar os sacrifícios que lhe foram pedidos na fase final deste projeto.

RESUMO

Uma tendência na indústria de software tem sido a de disponibilizar recursos para que os usuários finais possam configurar ou estender suas aplicações, numa tentativa de ampliar seu escopo de usabilidade e aplicabilidade e, assim, satisfazer as reais necessidades destes usuários. Infelizmente, a simples disponibilidade de tais recursos não capacita estes usuários a estender suas aplicações, pois, normalmente, falta-lhes conhecimento de como utilizá-los. Adler e Winograd [ADLER '92] propõem que o critério básico para a usabilidade de um software é a extensão do apoio que ele proporciona para que os usuários possam compreender e apreender o seu funcionamento, adaptação e extensão. A área de programação por usuários finais pode ser vista como uma resposta a tais anseios. Entretanto, que seja de nosso conhecimento, não existe um modelo teórico de consenso que apóie o desenvolvimento de aplicações extensíveis de forma satisfatória.

Neste trabalho, tomamos como base a visão do software como um artefato de meta-comunicação, proposta pela Engenharia Semiótica [DE SOUZA '93], e olhamos seu "uso" como um processo de comunicação entre o designer do software e o usuário. Partindo desta visão, empregamos o Modelo de Comunicação Verbal de Jakobson [JAKOBSON '60] para compreender o efeito das funções da linguagem sobre tal processo. Apoiados neste conhecimento e nos princípios do Contínuo Semiótico e da Abstração Interpretativa, desenvolvidos pela Engenharia Semiótica [DE SOUZA '01], apresentamos um modelo teórico para a tarefa de programação por usuários finais que respeita os critérios de usabilidade propostos por Adler e Winograd. Este modelo é composto de um processo para a realização de extensões ao software; de uma linguagem-tipo para programação por usuários finais, que emprega recursos comunicativos para a manutenção da coesão e coerência textual, tais como o uso de anáforas, metáforas e metonímias; e de uma arquitetura de software baseada em conhecimento com mecanismos específicos para apoiar a interpretação de instâncias desta linguagem-tipo e para realizar a manutenção dos princípios da Engenharia Semiótica anteriormente citados.

Palavras-chave: Programação por Usuários Finais, Linguagens de Extensão, Engenharia Semiótica, Interação Humano-Computador.

ABSTRACT

One of many noticeable trends in the software industry has been to create resources to configure or extend applications available to end-users, in an attempt to broaden the applications' usability and applicability, thus satisfying the real needs of these users. Unfortunately, simply making such resources available does not enable end-users to extend their applications, for they normally lack the knowledge of how to use them. Adler and Winograd [ADLER '92] propose that the key criterion for the usability of software should be the extent to which it supports the understanding and learning of its working, adaptation and extension. The area of End-User Programming could be seen as an answer to such demand. However, to the best of our knowledge, there is no agreement regarding a theoretical model that reasonably supports the development of extensible applications.

In this work we take the view of software as a meta-communication artifact, proposed in the Semiotic Engineering [DE SOUZA' 93], and cast its use as a communication process between the software designer and the end-user. Taking up this view, we make use of Jakobson's Verbal Communication Model [JAKOBSON '60] to understand the effect of the functions of language on this process. Supported by this knowledge and by the Semiotic Continuum and Interpretive Abstraction Principles, proposed by Semiotic Engineering [DE SOUZA' 01A], we present a theoretical model for the end-user programming task that pursues Adler and Winograd's proposal. This model is composed by a process for the creation of software extensions; a type-language for end-user programming that uses communicative resources to maintain the textual cohesion and coherence, such as the use of anaphors, metaphors and metonymy; and a software architecture with specific mechanisms to support the interpretation of instances of that type-language and to maintain the Semiotic Engineering principles previously cited.

Keywords: End-User Programming, Extension Languages, Semiotic Engineering, Human-Computer Interaction.

SUMÁRIO

Agradecimentos **III**

Resumo **IV**

Abstract **V**

Sumário **VI**

Lista de Figuras **IX**

Lista de Tabelas **XI**

Lista de Abreviaturas ou Siglas **XII**

Notação Utilizada neste Trabalho **XIII**

1. Introdução **1**

2. Fundamentação teórica **10**

1. A Engenharia Semiótica ***10***

2. Paradigmas atuais da tarefa de EUP ***17***

2.1. O paradigma de programação paramétrica 20

2.2. O paradigma de programação imitativa 25

2.3. O paradigma de programação descritiva 31

3. Um modelo semiótico para a tarefa de *EUP* **39**

1. O uso de software extensível como um processo comunicativo ***40***

1.1. O uso normal do software extensível 41

1.2. O uso do software extensível na criação de extensões 48

2. A natureza e perfil dos códigos no software extensível ***55***

2.1. O princípio da Abstração Interpretativa 57

2.2. O princípio do Contínuo Semiótico 61

4. Uma análise da linguagem de planos das pessoas **66**

1. Considerações gerais sobre a linguagem de planos das pessoas ***67***

2. Uma análise lingüística da linguagem de planos das pessoas ***69***
2.1. A composição da linguagem 69
2.2. A estrutura da linguagem 73

5. Uma linguagem-tipo para *EUP* 78

1. Aplicações extensíveis de software ***79***

2. Uma linguagem-tipo para EUP para o Modelo Semiótico ***87***
2.1. A definição de uma linguagem-tipo para *EUP* 90
2.1.1. Conceitos básicos de uma linguagem-tipo para EUP 91
2.1.2. O núcleo da linguagem 93
2.1.3. A metalinguagem 112
2.2 Sobre a expressividade das *EUPLs* derivadas da linguagem-tipo para *EUP* proposta 129

6. A organização da tarefa de *EUP* 132

1. O processo global de realização de uma extensão ***133***

2. A etapa de realização da extensão ***136***
2.1. O processo de criação de uma nova ação 136
2.2. O processo de criação de uma nova entidade 143
2.3. O processo de criação de uma relação 149
2.4. O processo de modificação de uma extensão 150
2.5. O processo de revogação de uma extensão 151

3. A etapa de validação sintática ***152***

4. A etapa de validação semântica ***153***

5. O abandono da tarefa de EUP ***155***

6. Comentários finais à organização do processo ***155***

7. Conclusões 158

Anexo I. Resultados dos teste com uma versão preliminar da linguagem-tipo para *EUP* 172

1. Descrição do ambiente ***173***

2. Módulo 1. Avaliação das "formulações naturais" de expressão de planos ***175***

3. Módulo 2. Avaliação da expressividade da EUPL ***175***

4. Módulo 3. Avaliação da produção de textos em EUPL ***176***

5. Análise dos testes com a versão preliminar da EUPL ***177***
5.1. Módulo 1 177
5.2. Módulo 2 178
5.3. Módulo 3 179

Anexo II. Exemplos do uso de uma instância da linguagem-tipo para *EUP* **182**

1. Exemplo de extensões a um E-mailer simples *182*

2. Exemplo de extensão a uma Agenda simples *185*

3. Observações gerais sobre o ambiente de extensões *188*

Referências Bibliográficas **191**

Glossário **200**

LISTA DE FIGURAS

FIGURA 1: *FRAMEWORK* DE ENGENHARIA SEMIÓTICA ESTENDIDO PARA INCLUIR A TAREFA DE *EUP*..........14
FIGURA 2: TIPOS DE CONFLITOS ENTRE A USABILIDADE DISPONIBILIZADA PELO DESIGNER NO SOFTWARE E A USABILIDADE DESEJADA PELO USUÁRIO PARA O SOFTWARE. 16
FIGURA 3: DIÁLOGO *OPTIONS* DO *MS WORD™*..........20
FIGURA 4: DESCRIÇÃO DA TAREFA DE CRIAÇÃO DE UMA MACRO PARA SALVAR UM ARQUIVO NO FORMATO *RTF* [DE SOUZA '01]..........28
FIGURA 5: USO DE UM COMPONENTE *OCX* QUE NÃO SEGUE OS PADRÕES DE INTERAÇÃO DA APLICAÇÃO. 36
FIGURA 6: UM EXEMPLO DE DIÁLOGO CRIADO NO *DIALOG EDITOR* DO *MS VBA™*..........37
FIGURA 7: TEXTO GERADO PELO *MS VBA™* EQUIVALENTE AO DIÁLOGO DA FIGURA 6..........37
FIGURA 8: MODELO DE COMUNICAÇÃO VERBAL DE JAKOBSON PARCIALMENTE INSTANCIADO PARA O CASO DE USO NORMAL DO SOFTWARE EXTENSÍVEL..........42
FIGURA 9: MODELO DE COMUNICAÇÃO DE JAKOBSON COMPLETAMENTE INSTANCIADO PARA O CASO DE USO NORMAL DO SOFTWARE EXTENSÍVEL..........44
FIGURA 10: MODELO DE JAKOBSON INSTANCIADO PARA O CASO DE USO DO SOFTWARE EXTENSÍVEL NA CRIAÇÃO DE EXTENSÕES AO PRÓPRIO SOFTWARE. 49
FIGURA 11: UM EXEMPLO DE UMA MENSAGEM DE ERRO QUE EMPREGA UM CÓDIGO QUE PERTENCEM AO NÍVEL DE IMPLEMENTAÇÃO DO SOFTWARE 58
FIGURA 12: MUDANÇAS NA *UIL* COMO RESULTADO DE UMA EXTENSÃO..........59
FIGURA 13: O PRINCÍPIO DA ABSTRAÇÃO INTERPRETATIVA..........60
FIGURA 14: O PRINCÍPIO DO CONTÍNUO SEMIÓTICO. 62
FIGURA 15: *EBNF* PARA A GRAMÁTICA DE ALTO NÍVEL DE UM TEXTO NA *EUPL*..........64
FIGURA 16: UM EXEMPLO DA ESTRUTURA DO TEXTO REQUERIDA PELO PRINCÍPIO DO CONTINUO SEMIÓTICO. 64
FIGURA 17: DESCRIÇÃO DA RELAÇÃO DE CONTINUIDADE SEMIÓTICA ENTRE A *UEL*, *UIL*, E A *EUPL* DE UM SOFTWARE EXTENSÍVEL. 65
FIGURA 18: ARQUITETURA DE SOFTWARE PROPOSTA NO *SERG* PARA SOFTWARE EXTENSÍVEL. 81
FIGURA 19: GRAMÁTICA DA SUB-LINGUAGEM DE REFERENCIAÇÃO DE OBJETOS DE UMA *EUPL*..........94
FIGURA 20: GRAMÁTICA QUE DESCREVE OS MECANISMOS DE CONTROLE DE UMA *EUPL*..105
FIGURA 21: DESCRIÇÃO DA ESTRUTURA DE UMA ENTIDADE NA *ADKB*..........113
FIGURA 22: DESCRIÇÃO DA ESTRUTURA DE UM ATRIBUTO NA *ADKB*..........114

Figura 23: Descrição da estrutura de uma interface na *ADKB*. 114
Figura 24: Descrição da estrutura de uma ação na *ADKB*. 115
Figura 25: Descrição da estrutura de uma relação na *ADKB*. 116
Figura 26: Gramática que descreve os mecanismos de metalinguagem de uma *EUPL*. 117
Figura 27: Exemplo de texto de uma extensão na *EUPL*. 127
Figura 28: Modelo do domínio para um cliente de *e-mail* simples. 127
Figura 29: Mapeamento semântico do corpo da ação especificada pela extensão da Figura 27. 128
Figura 30: Processo global de realização de extensões em um software extensível segundo o Modelo Semiótico proposto. 135
Figura 31: Descrição do processo de criação de novas ações em um software extensível. 137
Figura 32: Descrição do processo de criação de uma nova entidade. 143
Figura 33: Descrição do processo de definição de atributos para as entidades. 147
Figura 34: Processo abstrato de modificação de uma entidade do modelo do software. 151
Figura 35: Processo de teste da validade semântica de uma extensão ao software. 154
Figura 36: Uma configuração para um ambiente de extensão para o Modelo Semiótico proposto. 157
Figura 37: Descrição do ambiente de programação do robô Karezim. 174
Figura 38: Modelo conceitual de um *e-mailer* simples, conforme visto por um usuário final. 183
Figura 39: Tela principal de uma Agenda simples com mecanismo de extensão. 185
Figura 40: Modelo conceitual de uma Agenda simples, conforme vista por um usuário final. 186

LISTA DE TABELAS

TABELA 1: TABELA COM EXEMPLOS DOS TIPOS DE REFERENCIAS A OBJETOS VÁLIDAS PARA A GRAMÁTICA BÁSICA DE UMA *EUPL* INSTANCIADA DA LINGUAGEM-TIPO AQUI PROPOSTA. 94

TABELA 2: TABELA COM EXEMPLOS DE MAPEAMENTO SEMÂNTICA DAS REFERÊNCIAS DE UMA *EUPL* INSTANCIADA DA LINGUAGEM-TIPO AQUI PROPOSTA SOBRE ELEMENTOS DA *ADKB*. 102

LISTA DE ABREVIATURAS OU SIGLAS

ADKB: *Application Design Knowledge Base.*

DRT: *Discourse Representation Theory.*

EBNF: *Extended Backus-Naur Form.*

EUP: *End-User Programming*

EUPL: *End-User Programming Language.*

IA: Inteligência artificial.

KRL: *Knowledge Representation Language*

LN: Linguagem Natural.

LP: Linguagem de Programação.

PD: *Programação por Demonstração.*

UEL: *User Explanation Language.*

UIL: *User Interface Language.*

UILx: *Extensible part of the User Interface Language.*

NOTAÇÃO UTILIZADA NESTE TRABALHO

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, que se encontra no final do texto.

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

Capítulo 1

INTRODUÇÃO

O objetivo deste trabalho é a busca de uma forma de possibilitar aos usuários finais obter melhores aplicabilidade* e usabilidade de um software. Aplicabilidade aqui diz respeito às condições sob as quais um artefato (uma função, uma ferramenta, etc.) poderia ser usado e usabilidade é vista de acordo com os critérios definidos por Adler e Winograd [ADLER '92] conforme a citação abaixo:

> *The key criterion of a system's usability is the extent to which it supports the potential for people who work with it to understand it, to learn, and to make changes.*

Estes autores vão mais longe propondo que para um software atingir um alto grau de usabilidade é necessário que o designer do software o projete de modo que ele sustente o aprendizado por parte dos usuários em dois níveis:

1. Aprendizado através do qual o usuário vem a entender **como** e **porque** o software funciona; e

2. Aprendizado através do qual o usuário descubra **como adaptar** e **estender** a tecnologia de forma a satisfazer suas demandas e as contingências de seu trabalho.

Os autores também propõem que é necessário criar um processo de design que possibilite aos designers de software aprender como melhor abordar os problemas de usabilidade.

A área de programação por usuários finais[1] (*EUP*) [CYPHER '93A] [NARDI '93] [GOODELL '99] pode ser vista como a busca de uma resposta aos anseios expressos por estes autores no que diz respeito aos dois primeiros itens. A idéia central dos pesquisadores desta área é permitir aos usuários finais modificar suas aplicações para que estas sirvam melhor às suas necessidades e/ou estilos de trabalho. Estas idéias surgiram na academia e foram de certa forma incorporadas pela indústria de software, com maior ou menor sucesso, em produtos bem conhecidos como, por exemplo, os pacotes de aplicações *Office*™ da Microsoft® e *SmartSuite*™ da Lotus®.

Na abordagem adotada neste trabalho, um **usuário final** é todo usuário que apresente alfabetização computacional suficiente para operar uma aplicação de software para a realização de suas tarefas de modo eficaz. Assim, queremos deixar claro que, nesta abordagem, não consideramos necessário que o usuário tenha *a priori* qualquer conhecimento de programação ou do funcionamento interno de um sistema computacional.

Possibilitar a um usuário final personalizar suas tarefas certamente aumenta o potencial de aplicabilidade e de usabilidade de uma aplicação. Contudo, isto requer que o usuário possa realizar algum tipo de alteração na sua programação. Estas alterações de funcionalidade da aplicação são denominadas **extensões** e podem ser realizadas basicamente de duas formas: 1) através do uso da linguagem de interface da aplicação ou 2) através do uso de uma linguagem de extensão que opere embutida na aplicação.

Tanto a academia quanto a indústria de software têm, em diferentes medidas, tentado abordar ambos os caminhos. Exemplos de mecanismos de criação de extensões por meio da linguagem de interface do software são encontrados na academia, por exemplo, nas pesquisas em programação por demonstração (PD)[2] [CYPHER '93A] [MYERS '96] e, dentro do grupo de Engenharia Semiótica no qual essa pesquisa se insere, na Programação Via Interface

---

1 *End-User Programming.*

2 *Programming by Demonstration.*

[BARBOSA '99]. Já a indústria de software tem fornecido um conjunto amplo de mecanismos de customização ao usuário como, por exemplo, os diálogos de configuração de aplicações, os mecanismos para a gravação de macros (que agrupam uma seqüência de comandos realizada pelo usuário na interface da aplicação) e até mesmo mecanismos para PD como, por exemplo, no *AgentSheets®* [REPENNING '00A] e no *Stagecast Creator®* [SMITH '00]. Exemplos de mecanismos que permitem a criação por meio de linguagens de extensão embutidas na aplicação podem ser encontrados com grande facilidade na indústria de software representados pelas linguagens de macro tais como: *Hypercard™* [HYPERCARD '93], *Visual Basic™* [MSVBASIC '95] e *LotusScript™* [LOTUS '95], as quais possibilitam ao usuário editar as macros gravadas, ou mesmo, "programar" extensões às suas aplicações.

No entanto, é importante notar que, apesar da disponibilidade de linguagens de macro nas aplicações comerciais atuais, poucos usuários conseguem criar extensões de forma satisfatória nestas aplicações [BARBOSA '97A]. As razões para este fato podem ser encontradas quando analisamos o quanto estas aplicações satisfazem as necessidades de aprendizagem levantadas por Adler e Winograd.

Primeiramente, é importante salientar que possibilitar a um usuário leigo programar requer bem mais que adicionar uma linguagem de macro a uma aplicação já existente, como tem sido amplamente feito na indústria de software. O uso efetivo de uma linguagem de macro requer do usuário o **entendimento da lógica de funcionamento**[3] da aplicação na qual ela está embutida e dos elementos que compõem o software. Somente em posse deste conhecimento o usuário poderá referenciar corretamente os elementos cujo comportamento deseja alterar e assim criar sua própria extensão sem destruir a estrutura lógica original da aplicação concebida pelo designer do software. Esta afirmação encontra apoio nos trabalhos de Eisenberg e DiGiano [EISENBERG '95] [DIGIANO '95] que nos mostram que, através do desvelamento progressivo[4] do funcionamento interno da aplicação, é possível facilitar o entendimento do usuário sobre como alterá-la. O aprendizado da lógica de funcionamento da aplicação corresponde ao primeiro nível de aprendizagem descrito por Adler e Winograd.

Em vista disso, é legítimo imaginar que seja mais fácil estender uma aplicação mais simples que uma mais complexa e, portanto, que a indústria de software poderia criar aplicações mais

---

[3] O design *rationale*.

[4] *Progressive disclosure*.

simples que pudessem ser combinadas. Isto é o que propõem Smith e Susser [SMITH '92]. No entanto, o anseio e a criatividade dos usuários no uso de software, tentando realizar tarefas não projetadas para eles e, principalmente, a força do marketing, têm feito com que a indústria de software eleve o volume de funcionalidades das aplicações a cada nova versão, tornando-as cada vez mais complexas. Lamentavelmente, este aumento de funcionalidades vem em detrimento da compreensão do funcionamento das aplicações aumentando, deste modo, as dificuldades para a criação de extensões nestas aplicações.

Em segundo lugar, precisamos avaliar o custo acarretado pela inclusão de mecanismos para a programação por usuários finais nas aplicações. Se, por um lado, tal inclusão vem possibilitar a diminuição da necessidade de um grande número de novas funcionalidades nestas aplicações, por outro, para que um usuário possa fazer um correto uso destes mecanismos de extensão, é necessário o **entendimento do funcionamento do ambiente de extensão** que foi adicionado à aplicação para suportá-los. Além disso, tanto estes mecanismos quanto o ambiente devem possuir uma representação coerente dos elementos da linguagem de interface da aplicação que o usuário está acostumado a manipular, pois, em última instância, é esta linguagem que refletirá todas as alterações sofridas pelo software. Este conhecimento faz parte do segundo nível de aprendizagem discutido por Adler e Winograd. A ausência de mecanismos que facilitem a assimilação deste conhecimento provoca um aumento no grau de dificuldade para a criação de extensões nestas aplicações.

Por último, é relevante observar que não basta incluir uma linguagem e um ambiente de extensão às aplicações. É necessário que o usuário seja apoiado na **tarefa da criação de sua extensão**, uma vez que esta tarefa normalmente envolve a criação de novos elementos de interface para a aplicação ou a alteração de elementos já existentes, requerendo conhecimento mais avançado de software e hardware, conhecimento o qual os usuários finais não necessariamente possuem. A criação de elementos de interface não requer somente conhecimento de software e hardware, mas também das regras de design[5] empregadas pelo designer do software durante a construção da aplicação. Estas regras determinam o modo pelo qual os elementos de interface são organizados para criar a **linguagem única de interação** da aplicação [DE SOUZA '01]. Caso o usuário não siga estas regras, poderão ser criadas extensões que operem de modo totalmente incoerente com o restante da aplicação, o que poderá resultar na degradação da usabilidade da aplicação como um

[5] Parte integrante do design *rationale* da aplicação.

todo. Este tipo de conhecimento não é contemplado nas atuais aplicações comerciais de software tendo sido alvo de pesquisas na academia sobre o tema da lógica do design [MORAN '94]. Todos estes fatores dificultam ainda mais a realização da tarefa de extensão por parte dos usuários finais.

Os problemas expostos nos parágrafos anteriores nos mostram que, apesar de ter feito um esforço respeitável para melhorar a usabilidade de suas aplicações, a indústria de software ainda não abordou o problema de forma completa. A ausência, nas aplicações atuais, de mecanismos que facilitem ao usuário assimilar o conhecimento da lógica da aplicação, da linguagem e de seu ambiente de extensão e das regras de design empregadas na sua construção diminui não somente as chances de que este usuário consiga identificar oportunidades reais de extensão da aplicação, mas também pode levar a falsas oportunidades de extensão (quando o usuário, por não conhecer totalmente a aplicação, cria extensões para as quais os softwares já dispõem de funcionalidade). Além do mais, mesmo que este usuário consiga identificar corretamente estas oportunidades, ele, somente através da aplicação, não terá acesso a conhecimento, nem apoio para poder realizá-las.

Tais problemas, além de reafirmarem a proposta de Adler e Winograd da necessidade de mecanismos de auxílio à aprendizagem, salientam, em particular, a necessidade de um **mecanismo de explicação** que atue como facilitador da tarefa de aprendizagem contínua à qual os usuários finais estão submetidos nas aplicações extensíveis. Eles também nos indicam que a **dimensão de comunicação** passa a ter um papel preponderante na aplicabilidade e usabilidade do software, uma vez que ela é um elemento essencial nos processos de explicação e de aprendizagem. Estes problemas ainda mostram a necessidade da criação de ferramentas específicas que estejam intimamente integradas à arquitetura das aplicações extensíveis e que suportem a explicação da tarefa de *EUP* aos usuários finais. Para apoiar a criação de tais ferramentas, é necessário um **modelo da tarefa de *EUP*** que caracterize seus elementos e descreva sua organização no tempo e no espaço. Tanto quanto seja de nosso conhecimento não existem, nos trabalhos atuais da área de programação por usuários finais, modelos teóricos descritivos para a tarefa de *EUP*. A ausência de tais modelos impossibilita a realização de uma avaliação sistemática da qualidade dos mecanismos atualmente implementados e, também, o desenvolvimento de novas arquiteturas de software que implementem mecanismos para um suporte adequado à tarefa de *EUP*.

Neste trabalho, propomos um modelo para a tarefa de *EUP* baseado na **Semiótica** [ECO '76] [PEIRCE '31], mais especificamente na **Engenharia Semiótica** [DE SOUZA '93], e no **Modelo de Comunicação Verbal** de Jakobson [JAKOBSON '60]. A justificativa para tal escolha se

deve ao fato de que em Semiótica estudamos processos de representação e comunicação que podem explicar e prever fenômenos de *EUP*[6] e o Modelo de Comunicação Verbal de Jakobson nos fornece um *framework* para analisarmos os elementos e funções da linguagem envolvidos nos fenômenos de comunicação. Deste modo, adotaremos uma abordagem diferente para o desenvolvimento de software, que usa a perspectiva de que o computador é uma mídia. Tal perspectiva, proposta por Kammersgaard [KAMMERSGAARD '88] e por Andersen [ANDERSEN '93B], tem tido uma aceitação crescente na comunidade de IHC [DE SOUZA '00].

Em Engenharia Semiótica, entendemos uma aplicação extensível como uma mensagem única e unidirecional na qual o designer[7] está dizendo para o usuário:

- O espectro de problemas que a aplicação está apta a resolver — ou seja, qual o modelo de funcionalidade desta aplicação;
- Como é possível interagir com a aplicação para resolver estes problemas — ou seja, qual é o modelo de interação desta aplicação; e
- Quais as possibilidades de se estender esta solução — ou seja, qual o modelo de extensibilidade desta aplicação.

Uma vez que o software é visto como uma mensagem, o **uso do software** poderá ser interpretado como um **processo de comunicação e representação**, no qual o designer e o usuário ocuparão, respectivamente, os papéis de emissor e receptor da mensagem. Vista desta perspectiva, a tarefa de *EUP* é aquela em que o usuário está envolvido em dois papéis diferentes: o de usuário da aplicação propriamente dito, e o de designer — quando ele está realizando uma extensão à aplicação. É interessante observar que o usuário será um designer privilegiado, pois ele criará extensões para si mesmo. Este duplo envolvimento nos mostra como este processo de comunicação é complexo. Ele envolve a comunicação entre:

- O usuário — como usuário final — e o designer original: comunicação na qual o papel do designer é o de explicar o que é a aplicação e o que ela poderá ser através de extensões (elementos definidos pelo designer na construção da aplicação);

---

[6] Esta afirmação se deve ao fato de a realização de uma extensão envolver a criação de novos signos e novos significados na interface da aplicação. Visto que tais tarefas são um processo semiótico, o emprego das metodologias de pesquisa desenvolvidas na Semiótica pode auxiliar na compreensão e previsão dos fenômenos de significação e interpretação encontrados em tais tarefas.

[7] Neste trabalho, usaremos designer e designer de software como sinônimos.

- O usuário — como estendedor — e o ambiente de extensão: comunicação na qual o entendimento do ambiente de extensão é necessário para realizar de maneira completa a tarefa de *EUP* (este ambiente deve conter a mensagem do designer sobre as possibilidades de extensão e os diálogos possíveis e necessários para que elas se concretizem); e
- O usuário — como estendedor — e os demais usuários: quando algum usuário fizer uso da extensão que o usuário estendedor disponibilizou na sua aplicação (garantindo a usabilidade da aplicação por meio de sua interface).

A complexidade deste processo de comunicação advém da grande inter-relação existente entre as linguagens nele empregadas que compõem uma aplicação extensível, a saber, as linguagens de interface, de explicação e de extensão. O alto grau de inter-relação destas linguagens nos levou a adotar dois princípios da Engenharia Semiótica — os princípios da Abstração Interpretativa e do Contínuo Semiótico [DE SOUZA '01] —, os quais serão discutidos no Capítulo 3, como pressupostos básicos deste trabalho. Estes princípios permitem regular a forma que esta inter-relação deve ter. Eles também apontam para a necessidade de uma nova arquitetura de software, a qual será discutida no Capítulo 5. Esta arquitetura incorpora uma base de conhecimento que deve incluir um modelo do domínio da aplicação, e um modelo do software (que conterá as regras de design empregadas na construção do software). Esta base de conhecimento servirá de suporte aos mecanismos responsáveis pela manutenção dos dois princípios supra citados. Além disso, ela deverá ser empregada pelo mecanismo de geração de explicações para possibilitar explicações sensíveis aos elementos pragmáticos da tarefa do usuário.

Nossa visão do software como uma mensagem e do seu uso como um processo de comunicação e representação nos possibilitou identificar os recursos necessários à definição de uma **linguagem-tipo*** **para *EUP*** que leva em consideração os aspectos comunicativos necessários à melhor expressão dos usuários finais. Assim, uma linguagem de extensão que seja uma instância desta linguagem-tipo difere radicalmente das linguagens atualmente empregadas pela indústria de software e também das linguagens propostas na academia para o mesmo fim. Ela apresenta um mecanismo para referenciação a objetos que faz uso de figuras de linguagem, estruturas lingüísticas comumente usadas pelas pessoas, para facilitar a expressão de referências a objetos complexos. Sua estrutura foi definida por meio de uma análise lingüística da linguagem de planos que as pessoas comuns usam no seu dia-a-dia. Ela também incorpora mecanismos para que o usuário possa

expressar aspectos interativos e explicativos e agentes inteligentes de construção de interface que auxiliam o usuário no uso destes mecanismos.

É importante salientar que o modelo proposto neste trabalho trata somente de **sistemas adaptáveis**, ou seja, sistemas que permitem criar extensões a partir de uma intenção expressa pelo usuário. Não abordamos aqui os sistemas adaptativos, que inferem, através de técnicas de aprendizagem de máquina, a intenção do usuário e, a partir disso, criam extensões à aplicação automaticamente [GIRGENSOHN '92]. Isto implica que a aplicação não tem a capacidade de alterar a si própria e, portanto, que todas as mensagens que ela poderá gerar ou receber — isto é, que compõem sua funcionalidade (inclusive a sua capacidade de extensão) — já devem estar previstas no seu design original. Do mesmo modo, trataremos somente dos aspectos relativos às linguagens de extensão e à manutenção do Contínuo Semiótico entre as linguagens envolvidas nas aplicações extensíveis.

Assim, a contribuição central deste trabalho é a introdução de um modelo teórico para a tarefa de *EUP*, a que chamamos de **Modelo Semiótico**, baseado nos princípios da Abstração Interpretativa e do Contínuo Semiótico, provenientes da Engenharia Semiótica, e no Modelo de Comunicação Verbal de Jakobson. Este modelo é composto de um processo para a realização de extensões que é intimamente ligado a uma linguagem-tipo para *EUP* e é apoiado por uma arquitetura com mecanismos específicos para garantir a manutenção dos princípios da Engenharia Semiótica sobre os quais ele é construído, como será visto ao longo deste trabalho.

No Capítulo 2, apresentaremos alguns conceitos relacionados à Engenharia Semiótica e ofereceremos uma classificação para os paradigmas de programação por usuários finais atuais por nós desenvolvida. O núcleo teórico de nosso modelo é abordado no Capítulo 3. No Capítulo 4, apresentamos uma análise da linguagem de planos empregada pelas pessoas no seu dia-a-dia. Esta análise servirá de base para a construção da linguagem-tipo para *EUP* proposta em nosso modelo. No Capítulo 5 descreveremos a arquitetura de software necessária à manutenção dos princípios da Abstração Interpretativa e do Contínuo Semiótico e a linguagem-tipo para *EUP* baseada no nosso modelo, descrevendo sua sintaxe e sua semântica. O processo de construção de extensões que faz parte deste modelo é tratado no Capítulo 6. Finalmente, no Capítulo 7, comparamos nosso modelo aos paradigmas atuais para a tarefa de *EUP*, consolidamos nossas contribuições, discutimos o escopo desta abordagem e apontamos algumas direções para trabalhos futuros.

Acrescentamos ao texto central deste trabalho dois anexos. No primeiro, apresentamos os resultados dos testes realizados com uma versão preliminar de uma instância da linguagem-tipo proposta no Capítulo 5. Tais testes servirão para avaliar a qualidade da linguagem proposta e detectar falhas na sua usabilidade. No segundo, apresentamos alguns exemplos mais completos do uso de uma instância da versão da linguagem-tipo para *EUP* proposta neste trabalho.

C a p í t u l o   2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta alguns conceitos empregados neste trabalho. Em particular, descreveremos os conceitos da Engenharia Semiótica de modo a fundamentar o modelo teórico que desenvolveremos nos Capítulos 3, 5 e 6. E, em seguida, apresentaremos uma classificação para os paradigmas de programação por usuários finais atualmente em uso e discutiremos suas virtudes e fraquezas.

1. A Engenharia Semiótica

Muitos dos problemas enfrentados pelos usuários finais para a programação das aplicações extensíveis atuais são devidos às descontinuidades existentes entre sua interface e seu ambiente de extensão, mais especificamente entre as linguagens empregadas nestes dois ambientes. Tais descontinuidades são causadas:

1. Pela ausência de um mapeamento entre elementos empregados na linguagem de interface do software e na linguagem do seu ambiente de extensão ou vice-versa; ou
2. Pela falta de clareza no mapeamento existente entre estas duas linguagens.

Exemplos do primeiro caso podem ser reportados às situações em que a presença de um elemento de interface como, por exemplo, um ícone ou uma função de um menu, não tem sua expressão na linguagem do ambiente de extensão da aplicação ou vice-versa. Ou seja, apesar de o usuário poder comunicar sua intenção ao usá-lo na interface ele não poderá realizar esta mesma ação através da linguagem do ambiente de extensão ou vice-versa. Exemplos do segundo caso podem ser reportados às situações em que, apesar de o mapeamento entre as duas linguagens existir, não é possível ao usuário inferir diretamente quais elementos de uma linguagem estão sendo mapeados em que elementos da outra linguagem. Ou seja, o mapeamento entre as duas linguagens não é 1 para 1 e, além do mais, não é coeso, podendo um elemento de uma linguagem aparecer mapeado em várias partes de um elemento da outra linguagem e/ou podendo estas partes estarem em locais desconexos no texto* resultante.

Os dois tipos de situações citadas acima são muito similares às encontradas nas ocasiões em que duas pessoas que falam línguas diferentes estão tentando se comunicar para realizar uma tarefa. Para que a comunicação entre estas pessoas possa de fato ocorrer deverá haver uma forma de fazer uma tradução entre as duas línguas, o que normalmente é feito por um intérprete. No caso de uma aplicação extensível, estas duas entidades são o ambiente de uso do software (sua interface) e o seu ambiente de extensão e as duas línguas são a linguagem de interface e a de extensão. O problema todo está no fato de que o intérprete deveria referir-se centralmente ao modelo de usabilidade do software percebido pelo usuário. No entanto, na maioria das vezes, estes ambientes foram gerados de forma desconexa e seus modelos de usabilidade são diferentes, gerando, assim, problemas de comunicação pela falta de um intérprete competente.

Estes **aspectos comunicativos** da interação humano-computador podem ser estudados sob a luz da Semiótica Computacional [ANDERSEN '93A] [ANDERSEN '90] [DE SOUZA '93] [GUDWIN '97] [JORNA '96] [KAMMERSGAARD '88] [NADIN '88B] [NAKE '94] [NOTH '97] [DE OLIVEIRA '99]. Em particular, a investigação de tarefas de engenharia envolvidas no design e implementação de interfaces com os usuários, previamente exploradas dentro do *framework* da Engenharia Cognitiva [NORMAN '86], serão aqui abordadas sobre o prisma da Engenharia Semiótica [DE SOUZA '93].

A Teoria Semiótica [ECO '76] [PEIRCE '31] provê um *framework* para o estudo dos processos de significação e comunicação. Em Semiótica, um **signo*** é o produto de uma relação triádica entre o ***representamen**** (isto é, aquilo que representa alguma coisa para alguém), o **objeto**

(isto é, a coisa que ele representa) e seu **interpretante*** (isto é, um pensamento, uma sensação, uma ação ou um outro signo) [NADIN '88A]. O processo de interpretação de um signo é denominado de **semiose***. No entanto, o interpretante de um signo pode ser visto como um outro signo podendo, assim, ser interpretado em um número indefinido de camadas de significados, trazendo à mente uma variedade de outros signos e significados. Este processo de geração de uma cadeia de significados não tem limites por natureza e é denominado de **semiose ilimitada***, trazendo conseqüências críticas para a comunicação dentro dos modelos computacionais.

Tratar o software como um artefato de comunicação e representação pode trazer novos insights para a dualidade entre designer e usuário final dentro do campo de aplicações extensíveis. Um sistema de comunicação é usado para passar uma mensagem de um emissor a um receptor através de um meio*. Um meio pode aceitar mensagens em uma faixa específica de formas, assim, o emissor deve codificar a mensagem dentro desta faixa. O código* é usado para transmitir as intenções e significados do emissor ao receptor. Para expressar este fato, o emissor seleciona um conjunto de signos possíveis de acordo com o meio e atribui a eles alguma significação esperando que o receptor os interprete da mesma maneira [JAKOBSON '60].

A essência desta base Semiótica é que quando dois indivíduos se comunicam eles negociam os significados em uma conversação, de tal modo que a semiose ilimitada torna-se pragmaticamente* restrita a um território de compreensão mútua. A comunicação tem um sucesso neste nível quando, de alguma forma, os interpretantes destas pessoas convergem para configurações estáveis e reciprocamente compatíveis. Quando existem evidências de que esta convergência não foi atingida, as pessoas entram em um processo de conversação usando a linguagem para regular o significado da própria linguagem.

Quando olhamos o software como um artefato de comunicação e representação, ou seja, um signo, veremos que ele representa a mensagem envolvida no processo de comunicação entre o designer do software e o seu usuário. Esta mensagem refletirá o interpretante do designer sobre o domínio do problema para o qual o software foi construído (o objeto deste signo). No entanto, ela não necessariamente corresponderá ao interpretante atual do designer sobre este domínio, visto este estar em constante transformação devido ao processo de semiose ilimitada. Ela na verdade representa o **interpretante cristalizado do designer*** no momento da implementação do software. A observação deste fato é de suma importância para a compreensão deste processo semiótico como um todo. O fato de o designer não estar mais presente diretamente no diálogo tem

sérias implicações sobre a arquitetura do software, conforme será discutido no Capítulo 3. O problema todo ocorrerá quando o usuário não entender o funcionamento do software, ou seja, quando o seu interpretante não convergir para uma configuração estável e compatível com o do designer. Neste caso, ele somente terá disponível para dialogar, na busca de um significado comum, o **preposto do designer** — o próprio software. Logo, a implementação do software define os limites do diálogo de negociação de significados entre o usuário e o preposto do designer e o único recurso que o usuário terá agora para entender o significado que o designer está tentando comunicar são os vários textos codificados nas linguagens embutidas no software pelo designer.

De uma forma geral, a Engenharia Semiótica, como vista neste trabalho [DE SOUZA '93] [DE SOUZA '96A] [DE SOUZA '01], está interessada na caracterização semiótica do design, interpretação e uso de linguagens artificiais específicas tentando, assim, capturar o significado tecnológico das máquinas semióticas fictícias de Umberto Eco [ECO '88]. Em nossa abordagem, a interface da aplicação de software é vista como uma **mensagem única e unidirecional** do designer para os usuários. Mais ainda, visto que a interface também é capaz de trocar mensagens com o usuário, ela é vista como um **artefato de meta-comunicação** [DE SOUZA '93]. Esta mensagem única e unidirecional transmite para o usuário a resposta a duas questões fundamentais:

- Qual é a interpretação do designer sobre os tipos de problemas que a aplicação pode resolver? e,
- Como os usuários deveriam interagir com a aplicação para implementar as soluções para seus problemas, dada esta interpretação?

Logo, nesta abordagem, o uso de um software pode ser visto como um processo de comunicação em que o designer é o emissor da mensagem e os usuários, os receptores, conforme veremos em mais detalhes no Capítulo 3.

Para exemplificar este processo de comunicação, apresentamos na Figura 1 um esquema da nossa abordagem, onde o designer, o computador e o usuário exercem os papéis de emissor, de meio/canal de transmissão e de receptor, respectivamente. O designer, como emissor, criará uma mensagem — o software. O software projetado é a concretização do **significado pretendido** da mensagem do designer — isto é, a sua compreensão do problema, a solução por ele proposta e as possibilidades de extensão desta solução por ele vislumbradas [DE SOUZA '96A]. O conteúdo desta mensagem será decorrente de um **modelo conceitual abstrato** do software (proveniente da

análise do domínio do problema, das classes de usuários e das tarefas que se pretende apoiar) e das decisões de design e de implementação tomadas pelo designer, resultando em um **modelo de usabilidade projetado**. Deste modo, ele representa o interpretante cristalizado do designer e, conforme Prates *et al.* [PRATES '00], conterá as *affordances* pretendidas pelo designer na linguagem de interface do software.

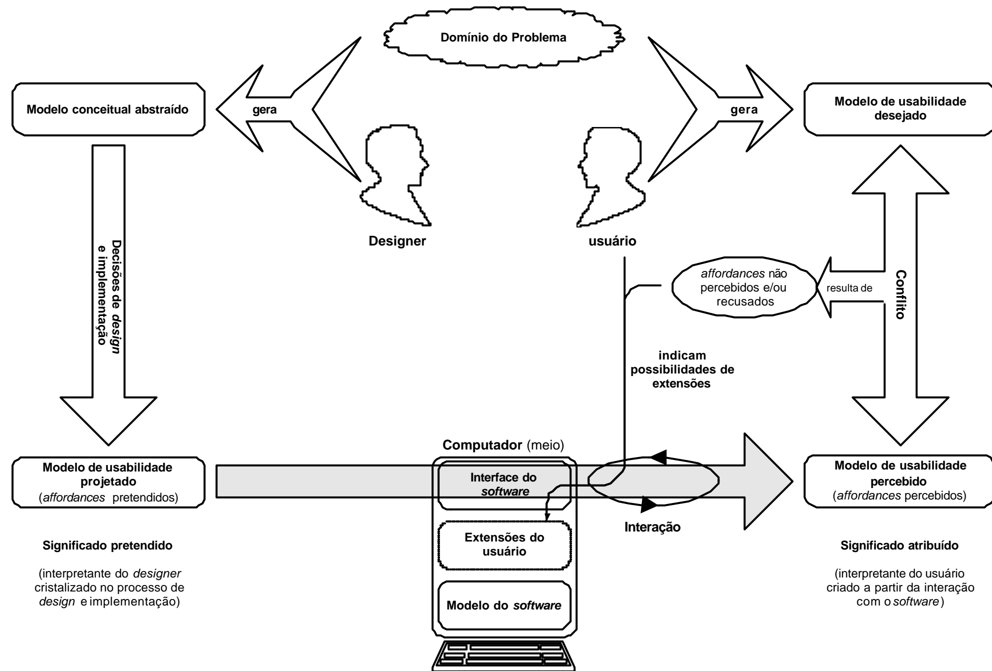

Figura 1: *Framework* de Engenharia Semiótica estendido para incluir a tarefa de *EUP*.

O usuário, por sua vez, através de seu contato direto com o domínio do problema, formará um **modelo de usabilidade desejado** para o software. Por outro lado, durante o processo de interação com a mensagem do designer — isto é, o software projetado — o usuário, por meio de sua interpretação dos elementos da interface deste software, atribuirá um significado a esta mensagem, criando assim o seu **modelo de usabilidade percebido** do software. Este é um modelo pessoal que representa o interpretante do usuário para a mensagem do designer. Ele contém as *affordances* percebidas pelo usuário, sendo, normalmente, composto de um subconjunto das *affordances* pretendidas pelo designer e pelo conjunto das *affordances* vislumbradas pelo usuário que não foram projetadas pelo designer. Assim, o **significado atribuído** por um usuário para o seu interpretante da mensagem do designer será uma variante (em diferentes graus de compatibilidade e consistência mútua) do significado atribuído por outros usuários, assim como daquele atribuído

pelo designer. Para a comunicação ter sucesso, todas estas variantes deverão ser compatíveis e consistentes entre si (sobretudo compatíveis e consistentes com a variante do designer).

Neste ponto, podem surgir problemas de usabilidade do software decorrentes dos conflitos entre o modelo de usabilidade desejado pelo usuário e o modelo de usabilidade percebido pelo usuário. Estes problemas, conforme descritos por Prates *et al.* [PRATES '00], poderão ser resultado da não percepção, pelos usuários, de *affordances* pretendidas pelo designer do software ou da recusa dos usuários em usá-las (por acharem que existe uma forma mais eficiente de executar uma determinada tarefa), como também é possível que o usuário venha a entender que a funcionalidade disponibilizada não é aquela por ele imaginada ou desejada. De Souza *et al.* [DE SOUZA '00] citam ainda que os problemas de comunicabilidade* podem ocorrem em três níveis, segundo os quais as *affordances* podem ser classificadas. São eles os níveis:

1. Operacional — relativo às ações individuais realizadas pelos usuários;
2. Tático — relativo a uma seqüência de ações que pode ser realizada para alcançar determinados objetivos; e
3. Estratégico — relativo as conceitualizações envolvidas na formulação dos problemas e no processo de solução de problemas.

Assim, os problemas de comunicabilidade resultantes da não percepção e da recusa dos usuários em usar as *affordances* pretendidas pelo designer indicam, por um lado, problemas no projeto do software mas, por outro lado, podem indicar também **oportunidades de extensão** ao software. Deste modo, por exemplo, no nível operacional, os usuários podem re-arranjar os signos disponibilizados pelo designer para enfatizar as *affordances* que são mais significativas para eles e, no nível tático, os usuários podem criar métodos customizados que melhor satisfaçam suas necessidades de interação [DE SOUZA '00] No nível estratégico, os usuários podem acrescentar à aplicação funcionalidades ausentes do modelo do designer que, contudo, fazem parte do domínio da aplicação. É relevante notar que a não-percepção e a recusa do uso das *affordances* pretendidas pelo designer também podem levar à criação de falsas extensões e de sinonímias, pois o usuário poderá criar extensões para funcionalidades já existentes.

É importante observar que estes conflitos podem pertencer a quatro classes distintas, conforme nos mostra a Figura 2. No caso 1, teremos as situações em que a usabilidade disponibilizada pelo designer abarca somente parte daquela desejada pelo usuário. Neste caso

poderá ser possível resolver o conflito de interesses por meio da criação de extensões ao software que complementem a parte ausente. No caso 2, teremos situações em que a usabilidade disponibilizada pelo designer está totalmente inclusa na desejada pelo usuário, mas não a completa. Neste caso, também poderá ser possível resolver o conflito por meio da criação de extensões que complementem a parte ausente. É importante salientar que nos casos 1 e 2 os conflitos somente poderão ser eventualmente resolvidos por extensão se o designer possibilitar ao usuário criar extensões que vão na direção de seus anseios, caso contrário não haverá solução para tais conflitos. No caso 3, teremos situações em que a usabilidade disponibilizada pelo designer e a desejada pelo usuário são totalmente desconexas. Aqui, a melhor solução para o conflito é o re-projeto total da aplicação, pois não haverá a possibilidade de se resolver o conflito com extensões à aplicação. No caso 4, teremos as situações em que a usabilidade disponibilizada pelo designer é superior à desejada pelo o usuário. Neste caso, não haverá necessidade da criação de extensões ao software, pois o mesmo já faz mais que o desejado. No entanto, será interessante que o designer preveja formas de possibilitar ao usuário desativar as funcionalidades não utilizadas do software por meio de mecanismos de customização. A existência de tais mecanismos permitirá ao usuário ajustar o software para as suas reais necessidades eliminando, assim, parte da carga cognitiva resultante do volume excessivo de funcionalidades disponibilizadas.

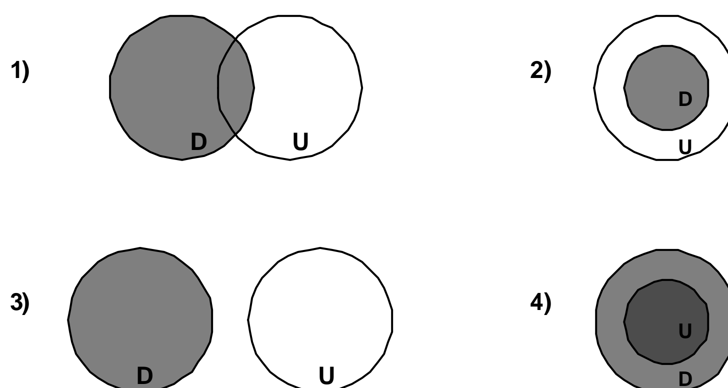

Figura 2: Tipos de conflitos entre a usabilidade disponibilizada pelo designer no software e a usabilidade desejada pelo usuário para o software.

Assim, nos casos em que é possível resolver os conflitos, o usuário poderá tomar o papel de designer e acrescentar à aplicação novos elementos de funcionalidade por meio da criação de extensões. A fim de realizar esta tarefa o usuário deverá conhecer o modelo da aplicação de forma a construir suas extensões de acordo com este modelo sem destruir a mensagem original do designer.

Este trabalho adota a abordagem da Engenharia Semiótica para o estudo do desenvolvimento de interfaces de software e da tarefa de programação por usuários finais.

2. Paradigmas atuais da tarefa de *EUP*

Conforme citado no Capítulo 1, a programação por usuários finais tem sido vista como uma solução para o problema de melhorar a usabilidade e aplicabilidade do software. Entretanto, estabelecer qual o melhor modelo para a tarefa de *EUP* a ser empregado nestas aplicações ainda carece de consenso. Uma revisão na literatura nos mostra que várias designações são empregadas para definir a capacidade de um usuário final alterar um software para que ele satisfaça suas reais necessidades, dentre as quais podemos citar: customização, parametrização, adaptação, programação por usuário final, extensão, personalização e *tailoring* Neste trabalho, usaremos a palavra extensão como um termo genérico para designar esta capacidade. Além disso, verificamos também que não existe uma classificação para os métodos e mecanismos empregados na tarefa de *EUP* que apresente uma base teórica sólida de modo a auxiliar o designer de software na identificação e solução de problemas que surgem na implementação de tais mecanismos.

Encontramos no trabalho de Trigg *et al.* [TRIGG '87] a identificação de quatro formas de adaptar um software, são elas:

1. A flexibilização — a disponibilização pelo designer do software de objetos e comportamentos que podem ser interpretados e usados diferentemente;
2. A parametrização — a escolha entre comportamentos alternativos previamente disponibilizados pelo designer no software;
3. A integração — a conexão de novos componentes, externos ou internos, ao software existente; e
4. O *tailoring* — a alteração do próprio software pela construção de aceleradores, comportamentos especializados e/ou a adição de funcionalidade.

Já Fischer e Girgensohn [FISCHER '90] identificam quarto possíveis formas de modificação de um software existente por usuários finais, são elas:

1. A definição de parâmetros (como em um diálogo de opções);
2. A adição de funcionalidade a classes de objetos existentes;

3. A criação de novas classes de objetos pela modificação de objetos existentes; e
4. A definição de nova funcionalidade a partir do zero.

É interessante observar que a primeira forma de Fischer e Girgensohn é equivalente à segunda forma de adaptação de Trigg *et al* e que as forma de 2 a 4 de Fischer e Girgensohn enquadram-se na quarta forma de adaptação de Trigg *et al.* (*tailoring*). Assim podemos ver que, Fischer e Girgensohn não consideram a integração de novos componentes diretamente como uma forma de alteração do software por usuários finais.

Cypher agrupa as abordagens para a programação por usuários finais em quatro categorias [CYPHER '93B], são elas o uso de:

1. Preferências — uso de alternativas predefinidas disponibilizadas pelo designer do software para acomodar as várias necessidades dos diferentes tipos de usuários, também chamada de parametrização;
2. Linguagens de roteiro[8] — uso de linguagens de programação pequenas e simples, cujo vocabulário é feito sob medida para os objetos e ações de um domínio particular de aplicação, na confecção de extensões;
3. Gravadores de macros — uso de mecanismos que possibilitam ao usuário gravar a seqüência de ações que ele realiza e agrupá-las sob um mecanismo de ativação da repetição da seqüência; e
4. Programação por demonstração — é uma elaboração sobre a gravação de macros que podem fazer uso de mecanismos de inferência para realizar a criação de programas generalizados a partir de seqüência de ações gravadas.

Apesar de Cypher apresentar uma classificação para os mecanismos utilizados na tarefa de *EUP*, ele não emprega um embasamento teórico mais rígido na sua organização. Portanto, sua classificação pouco auxilia os designers no seu trabalho de projetos de tais aplicações.

Mørch [MØRCH '97] apresenta um trabalho mais interessante ao propor uma classificação com um embasamento teórico na Engenharia Cognitiva, relacionando a distância de uso (existente

[8] *Script.*

entre os objetos de apresentação e o usuário) e a distância de design (existente entre os objetos de apresentação e o texto da implementação) a três possíveis formas de *tailoring*[9]:

1. Customização — modificação da aparência dos objetos de apresentação ou edição dos valores de seus atributos selecionando entre um conjunto de opções de configurações predefinidas;
2. Integração — adição de novas funcionalidades ao software através da ligação de componentes predefinidos dentro ou através de aplicações sem, no entanto, acessar o texto da implementação; e
3. Extensão — adição de novas funcionalidades às aplicações e aos componentes textuais das aplicações, em "pontos abertos" — isto é, áreas da implementação predefinidas pelo designer que podem apresentar diferentes alternativas de design.

Conforme podemos ver, a maioria destas classificações não apresenta um tratamento teórico sobre o problema da tarefa de *EUP*. Elas, em geral, descrevem as possíveis formas e mecanismos para alterar um software existente sem, contudo, criarem um modelo teórico que auxilie o designer na identificação e solução dos problemas que surgem na implementação de mecanismos que realizam a tarefa de *EUP*. Exceção deve ser feita à classificação de Mørch que emprega a Engenharia Cognitiva como base teórica. No entanto, como esta abordagem não leva em conta os aspectos comunicativos de cada mecanismo, que são o foco principal deste trabalho, ela não será utilizada neste trabalho.

Os mecanismos atuais de criação de extensões empregados por aplicações extensíveis podem ser classificados por meio dos paradigmas de computação por eles utilizados. Cada paradigma expressa, na verdade, o tipo de linguagem empregada pelos mecanismos que o implementam para comunicar suas possibilidades de extensão. Deste modo, podemos classificar estes paradigmas como paradigmas de programação: paramétrica, imitativa e descritiva. A seguir, apresentamos uma revisão destes paradigmas e dos principais mecanismos atualmente empregados na tarefa de extensão de aplicações. Nesta revisão, empregaremos a Engenharia Semiótica como base teórica de análise e apresentaremos uma crítica dos aspectos comunicativos destes paradigmas, procurando identificar o alcance que a tecnologia atual de aplicações extensíveis realmente proporciona para permitir a um usuário final criar suas próprias extensões.

---

[9] Mørch usa o termo *tailoring* como um nome geral para tarefa de alteração de um software já existente.

2.1. O paradigma de programação paramétrica

Neste paradigma de programação, a tarefa de *EUP* é comparável à tarefa de "seleção e ativação de elementos em um quadro de distribuição". Assim, o designer de software deverá disponibilizar para o usuário final, por meio da arquitetura da aplicação, um conjunto básico de módulos e uma forma de configurar estes módulos. Em vista disso, o usuário terá um conjunto fechado (formado contextualmente por um critério de coesão) de possibilidades de personalização[10] de tipos de tarefas sobre os quais as alternativas operam. Por exemplo, *View*, *General*, *Edit*, etc. na Figura 3 podem ser tarefas configuráveis por meio da manipulação de parâmetros a elas associados. Tais parâmetros podem ser mutuamente exclusivos ou interdependentes.

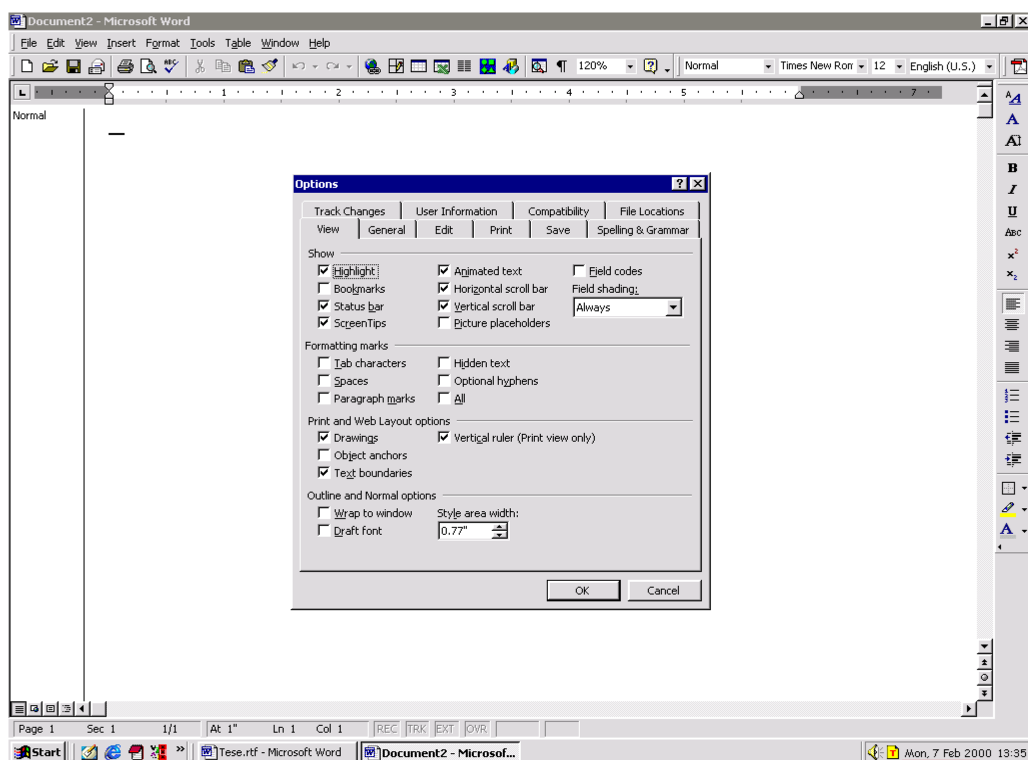

Figura 3: Diálogo *Options* do *MS Word*™.

Uma extensão à aplicação neste paradigma de programação (um novo significado a ser criado, ou um novo estado de coisas, no software) será, na maioria das vezes, resultante da composição da opção pela presença ou ausência de elementos primitivos fornecidos pelo designer ou pela variação de valores destes elementos. Então, neste caso, a extensão terá uma semântica eminentemente composicional. Caso o usuário possa indicar símbolos arbitrários como valores de

[10] *Customization*.

parâmetros (por exemplo, caso ele possa definir o nome de elementos como arquivos, diretórios, e similares), a semântica da extensão será denotacional — para todo símbolo que denota o nome ou valor de um elemento no modelo da aplicação ou do ambiente em que ela é executada. Em ambos os casos, o usuário estará mudando a semântica instanciada inicial da mensagem do designer e, portanto, estará realizando um tipo restrito de programação.

Existem dois níveis possíveis de atuação do usuário na criação de uma extensão à aplicação neste paradigma de programação. No primeiro, o usuário está limitado a selecionar as alternativas e a mudar os valores para os parâmetros disponibilizados pelo designer do software e, por conseguinte, ele não estará verdadeiramente criando uma extensão à aplicação, mas somente configurando-a. No segundo nível, entretanto, o usuário pode criar novos itens léxicos para serem apresentados na interface da aplicação, por meio da seleção, ativação e agrupamento de um conjunto de parâmetros e sua posterior associação a um nome. Os tipos de itens léxicos que poderão ser criados pelo usuário são previamente definidos pelo designer do software. Um exemplo deste fenômeno é a criação de estilos no *MS Word*™ para a definição da formatação de parágrafos para uma tese. Assim, um usuário poderá criar, por exemplo, os estilos: "Título da Capa", "Título do Capítulo" e "Bibliografia". Estes nomes compõem elementos léxicos que aparecerão no diálogo de interface que determinará o estilo de um parágrafo ou linha de texto. Os estilos que eles denominam agrupam um conjunto de valores para os parâmetros que determinam o alinhamento dos parágrafos e o estilo de fonte usado em cada caso. O usuário poderá ainda associar um destes estilos a um ícone que será disponibilizado numa barra de ferramentas[11] para acesso rápido. Neste caso, ele estará inserindo outro elemento léxico na interface. No entanto, a possibilidade de alterar estes elementos precisa ter sido antecipada pelo designer do software na arquitetura do software.

Um sério problema do ponto de vista da Engenharia Semiótica [DE SOUZA '93] pode ocorrer com alguns mecanismos que seguem este paradigma de programação — a possibilidade de **destruição da mensagem original do designer**[12]. Isto poderá ocorrer quando o designer de software deixar uma grande abertura nas possibilidades de alteração da representação das funcionalidades na linguagem de interface da aplicação. Assim, caso o usuário possa trocar os

[11] *Toolbar*.

[12] É interessante ressaltar que, pela abordagem da Engenharia Semiótica, a mensagem original do designer é uma mensagem unilateral que vai do designer para o usuário. Logo, o que temos aqui é um processo de comunicação com a ausência física do emissor onde o designer exercerá um papel semelhante ao do autor de um livro.

elementos de representação na linguagem de interface de forma aleatória como, por exemplo, mudar o ícone de uma barra de ferramenta associada a uma determinada funcionalidade do software por uma imagem aleatória qualquer, ou trocar o item lexical de um menu por uma outra palavra qualquer ou, em uma situação extrema, trocar o item lexical por uma palavra que represente uma outra funcionalidade do software, é possível que ele realize mudanças na linguagem de interface para as quais ele mesmo, após algum tempo, não consiga mais interpretar o significado dos novos signos. Por conseguinte, ele poderá perder a compreensão do modelo de usabilidade da aplicação.

É importante observar que a ordem de seleção dos parâmetros não tem valor semântico neste paradigma de programação. Em vista disso, este paradigma atende a um princípio de formação combinatório não-ordenado atuando, caracteristicamente, no nível da semântica lexical do discurso. Logo, para cada diferente possibilidade de personalização da aplicação neste paradigma de programação, o designer de software terá que disponibilizar um novo parâmetro e seu conjunto de alternativas para que o usuário possa utilizá-la. Assim sendo, a usabilidade dos mecanismos que seguem este paradigma de programação será determinada pelo número de parâmetros e suas alternativas disponibilizadas aos usuários nos diálogos de configuração. Todavia, um número elevado de alternativas, mesmo que contextualmente agrupadas, como ocorre na Figura 3, tem como conseqüência direta uma maior dificuldade de identificação da ação resultante de uma escolha individual. Então, o maior desafio neste paradigma de programação é fazer com que um usuário consiga prever o resultado da combinação de parâmetros sobre o modelo de usabilidade original da aplicação, frente a um número significativo de opções.

Este paradigma de programação é amplamente utilizado comercialmente sendo que, dentre os seus mecanismos de implementação mais adotados, podemos citar a configuração de preferências, a personalização da aparência[13] da interface e o uso de moldes[14].

A configuração de preferências

Este mecanismo possibilita ao usuário ativar ou desativar opções em um conjunto de parâmetros contextualizados que irão afetar o comportamento global da aplicação como, por exemplo, o diálogo de *Options* do *MS Word*™ apresentado na Figura 3. Este é, com certeza, o

---

[13] *Look-and-feel.*

[14] *Templates and Styles.*

mecanismo mais comumente usado que segue o paradigma de programação paramétrica. Entretanto, ele apresenta dificuldades de usabilidade para os usuários finais, visto que, apesar de eles estarem acostumados aos elementos da "linguagem interativa" empregada na interface da aplicação, eles geralmente não conseguem estimar qual é o efeito final de um conjunto de escolhas de parâmetros sobre o comportamento da aplicação. Isto ocorre devido ao fato de os usuários não apresentarem conhecimento suficiente do modelo de usabilidade da aplicação. Estes problemas se agravam pelo fato de que, na maioria das vezes, os itens lexicais empregados para esclarecer a função de um parâmetro não se relacionam com o domínio da aplicação, mas sim com as escolhas de implementação das funcionalidades do software feitas pelo designer.

É importante observar que, neste mecanismo, o usuário se restringe à seleção da combinação de *tokens* pré-definidos pelo designer, isto é, por meio deste mecanismo ele não consegue criar novos *tokens* na interface da aplicação. Assim sendo, o usuário não produz uma extensão verdadeira na aplicação, mas somente ativa ou desativa elementos, ou traços de elementos, que já compõem a aplicação. Em vista disso, podemos dizer que este mecanismo deixa bastante a desejar quanto a sua capacidade de permitir aos usuários finais adaptar suas aplicações.

A personalização da aparência da interface

Este mecanismo possibilita ao usuário realizar mudanças nos elementos visuais, ou léxicos (barras de ferramentas e menus) presentes na interface da aplicação. Isto é possível através da alteração de sua composição e/ou posição como, por exemplo, a barra de ferramentas que aparece à direita na Figura 3, cuja posição padrão abaixo do menu foi alterada. Seu grau de dificuldade de utilização é semelhante ao do mecanismo anterior. Entretanto, neste mecanismo, o usuário terá que inferir quais as mudanças locais válidas de serem realizadas na interface, enquanto no anterior ele tinha que escolher entre um conjunto amplo de alternativas que alteravam o comportamento global da aplicação.

Contudo, seus problemas são de outra natureza, no que diz respeito ao grau de liberdade permitido nas alterações locais. Por exemplo, algumas aplicações permitem aos usuários alterar os itens léxicos presentes na sua interface por outros itens quaisquer (como ocorre no *MS Word*™ a partir da versão 8). Do ponto de vista da Engenharia Semiótica, a possibilidade de realizar este tipo de alteração abre espaço para a **destruição da mensagem original do designer**, visto que o usuário estará trocando aleatoriamente o *representamen* de um signo da interface previamente escolhido pelo designer do software para gerar um determinado interpretante. A troca aleatória

deste *representamen* normalmente resulta na geração de um interpretante equivocado para o signo em questão, o que acarreta na perda da compreensão do modelo de usabilidade da aplicação pelos usuários. Deste modo, para evitar tal problema, todas as alterações locais realizadas por este mecanismo deverão ser expressáveis via um **perfil do usuário**[15] (isto é, por meio de uma configuração pessoal do usuário), de modo a garantir a manutenção do significado da mensagem original do designer. Logo, se levado a extremos, este mecanismo não é uma forma segura de personalização da aplicação.

Por outro lado, este mecanismo possibilita ao usuário o uso de novos elementos visuais ou lexicais para representar funções já existentes na aplicação. Assim, diferentemente do mecanismo anterior, o usuário pode criar novos *tokens* para a linguagem de interface da aplicação, porém não pode ainda criar novas funcionalidades e, portanto, também não pode criar extensões verdadeiras à aplicação. Conseqüentemente, o ganho que este mecanismo apresenta é a possibilidade de personalizar a linguagem de interface podendo-se, assim, corrigir algumas discrepâncias criadas pelo designer em relação à linguagem empregada pelos usuários para expressar os elementos do domínio da aplicação.

É importante ressaltar que, normalmente, o uso deste mecanismo é feito conjuntamente ao de outros como, por exemplo, do mecanismo de gravação ou edição de macros — que são mecanismos que pertencem aos paradigmas de programação imitativo e descritivo, respectivamente. Este funcionamento conjunto permite ao usuário criar extensões às aplicações, por meio dos outros mecanismos, e atribuí-las a um novo elemento representacional na interface, por meio deste mecanismo. Assim, esta operação conjunta possibilita a criação de extensões reais à aplicação. Entretanto, o emprego deste mecanismo sozinho não apresenta a adaptabilidade que queremos para uma aplicação extensível.

O uso de moldes

Os moldes possibilitam ao usuário agrupar um ou mais conjunto(s) de parâmetros (para um tipo específico de tarefa), dar uma denominação a este agrupamento e empregá-lo para customizar o leiaute[16] de um documento como um todo. Em algumas aplicações estes conjuntos, ou parte dele,

---

[15] *Profile do usuário.*

[16] *Layout.*

podem receber nomes individuais como estilos[17], *Shapesheets*, etc., podendo também ser aplicados individualmente a partes do documento com o qual se está trabalhando.

Este mecanismo está no limiar entre o paradigma de programação paramétrico e o imitativo. Como nos mecanismos que seguem a programação paramétrica, a seqüência de ações durante a criação de um molde, geralmente, não é significativa. Entretanto, o uso de moldes possibilita a criação de novos *tokens* (com novos significados) na aplicação como nos mecanismos que seguem a programação imitativa, que será vista na próxima seção.

Uma das vantagens deste mecanismo é que ele não se restringe a alterações globais na aplicação, admitindo a diferenciação e o compartilhamento de personalizações entre diferentes documentos e usuários. Isto facilita o uso eficiente de uma mesma ferramenta (extensão) por pessoas com perfis diferentes. A restrição de seu escopo de atuação constitui seu ponto forte, pois possibilita aos usuários um ajuste mais fino e individualizado na sua forma de trabalho do que os demais mecanismos que seguem a programação paramétrica. Assim, sua usabilidade esta associada à facilidade de criação e manipulação dos moldes.

2.2. O paradigma de programação imitativa

Neste paradigma de programação, a tarefa de *EUP* é comparável à "criação de uma réplica de um objeto". No entanto, neste caso, o objeto será um processo — uma seqüência de ações realizadas pelo usuário. Em vista disso, o designer de software deverá disponibilizar para o usuário final, por meio da arquitetura da aplicação, mecanismos que permitam ao usuário tanto definir qual seqüência de ações ele deseja replicar quanto ativar a replicação desta seqüência. Assim, o usuário terá um conjunto mais amplo de possibilidades de personalização da aplicação que no paradigma anterior, visto que ele poderá replicar quase todas[18] as seqüências de ações que podem ser realizadas por meio da linguagem de interação da aplicação.

Uma extensão à aplicação neste paradigma será resultante do conjunto de elementos primitivos (os itens léxicos fornecidos pelo designer) selecionados pelo usuário e da ordem em que ele realiza sua escolha. Deste modo, este paradigma atende a um princípio de formação combinatória ordenada, de modo que sua semântica será eminentemente procedimental.

---

[17] *Styles*.

[18] Normalmente, não é possível ao usuário utilizar ações que tenham relação direta com o mecanismo de aquisição da seqüência de ações.

Do ponto de vista lingüístico, quando mudamos do paradigma de programação paramétrica para o de programação imitativa, ocorre uma passagem do nível léxico para o nível sintático, visto que a ordem em que um conjunto de símbolos (itens lexicais) é encadeado passa a importar no significado final. Em outras palavras, passa-se do nível de um conjunto de símbolos (justaposição lexical) para o de uma estrutura de símbolos (sentenças). Esta mudança do nível gramatical do texto aumenta o poder de expressão da linguagem disponível ao usuário para descrever suas extensões, uma vez que se torna possível, a partir de um conjunto fixo de elementos lexicais (as opções disponíveis), obter um conjunto infinito de sentenças (as configurações finais). Além de aumentar o poder de expressão da linguagem, a diminuição do número de elementos léxicos traz vantagens significativas para o usuário final do ponto de vista cognitivo, uma vez que ele terá que aprender e relembrar o significado de um conjunto menor de opções. Esta observação está de acordo com os resultados da psicologia cognitiva que nos demonstra que o ser humano memoriza melhor estruturas do que itens independentes [PREECE '94].

Por criar réplicas a partir de uma seqüência de ações especificada por meio de interações na linguagem de interface, os mecanismos que seguem este paradigma de programação, normalmente, apresentam grandes dificuldades para representar operações condicionais, iterativas e também o conceito de variáveis. As soluções comumente encontradas para este problema recaem sobre dois casos:

1. O deslocamento da responsabilidade desta especificação para o usuário (isto é, ele deverá determinar onde devem existir condicionais, repetições e o que deverá ser variável); e
2. O uso de técnicas de inteligência artificial para extrair os possíveis elementos a serem generalizados da seqüência gravada [CYPHER '93] [MYERS '00].

Todavia, para que o usuário possa atuar de forma eficiente, ele terá ainda que aprender o significado das diferentes combinações de opções, ou seja, ele terá que inferir, a partir dos padrões de uso dos itens lexicais na linguagem de interação da aplicação, a gramática que define as combinações válidas de opções. Esta não é uma tarefa simples, devido às inúmeras possibilidades de combinações prováveis de serem realizadas numa aplicação comercial. Por conseguinte, a usabilidade dos mecanismos que se enquadram neste paradigma de programação é determinada pelo grau de facilidade apresentado na aprendizagem da gramática da sua linguagem de extensão. E, assim, o maior desafio deste tipo de paradigma de programação é disponibilizar uma forma

eficiente de transmitir ao usuário a gramática que controla os padrões de interação da linguagem de interface e de extensão.

Dentre os mecanismos mais importantes que seguem este paradigma de programação podem ser citados os gravadores de macros, que são amplamente empregados comercialmente, e a programação por demonstração ou por exemplos [CYPHER '93A], que por ser uma proposta mais avançada encontra-se menos utilizada comercialmente, tendo como seus atuais expoentes comerciais o *Stagecast Creator*® [SMITH '00] e o *AgentSheets*® [REPENNING '00].

A gravação de macros

Este mecanismo possibilita aos usuários: uma forma de "armazenar o histórico" de um conjunto de ações realizadas na interface da aplicação assim como associar um nome e um mecanismo de ativação (menu ou atalho[19] de teclado) a este histórico, e depois invocá-lo através do mecanismo de ativação especificado.

É interessante notar que o que qualquer macro está tentando fazer é promover a objetos de primeira-classe a dimensão temporal de um programa — isto é, "um trecho instanciado de processamento". Como apontado por Gelernter e Jagannathan [GELERNTER '90], as linguagens de programação não fazem normalmente este tipo de coisa, já que elas trabalham somente na dimensão estrutural do programa. O mecanismo de continuações[20], presente na linguagem de programação *Scheme* [ABELSON '93], é um passo nesta direção, uma vez que ele armazena a computação (isto é, a descrição restante da função e o ambiente em que ela ocorre) que falta ser processada a partir de um determinado ponto e possibilita voltar a este ponto para terminar o seu processamento posteriormente.

Por outro lado, é importante observar que as macros não atuam exatamente como uma réplica, pois elas transformam em variável um dado arbitrário de entrada. Assim, o principal problema com este mecanismo é que, devido ao fato de trabalhar na dimensão temporal, na maioria das vezes, as macros gravadas carregam consigo partes irrelevantes ou até mesmo indesejáveis do contexto histórico em que ocorreram. Durante a gravação, são armazenados todos (menos um, que é arbitrariamente tomado como dado de entrada) os *tokens* da linguagem de interface que foram adotados pelo usuário (conscientemente ou não) durante a fase de criação da macro.

---

[19] *Shortcut.*

Para melhor esclarecer o problema mencionado segue a análise de um exemplo. Vamos supor que há um usuário do *MS Word*™ que trabalhe em um grupo cujos integrantes operam em plataformas de hardware diferentes. Para possibilitar a troca de documentos entre os integrantes do grupo, definiu-se que o formato de arquivo *RTF* seria o padrão para todo documento a ser trocado. Por isto, este usuário deseja criar uma macro que lhe permita salvar seus arquivos diretamente em formato *RTF*, imitando o diálogo típico da aplicação (o "**Salvar como…**"). Sua tarefa de criação da macro compõe-se das etapas listadas na Figura 4.

Com a macro gravada, o usuário passaria a usá-la para salvar sistematicamente seu trabalho de grupo em formato *RTF*. A interface para a nova função seria o atalho de teclado que associou à macro. Da primeira vez em que ele realizasse estes passos, o usuário receberia uma mensagem de sistema perguntando se ele quereria sobrescrever o arquivo EXEMPLO.RTF — o nome do arquivo ativo quando da criação da macro. A necessidade de sempre responder que "não" e de fornecer um nome específico para a versão *RTF* (que não seria automaticamente presumido como a combinação do nome do arquivo corrente em edição seguido da extensão "*RTF*") o levaria a desistir de usar a macro. Ele teria que realizar quase sempre o mesmo conjunto de ações que realizava antes da sua criação, o que justamente tentara abreviar por meio da criação da macro.

1. Ativar o menu **Tools > Macro > Record new macro…**
2. Atribuir um nome para a macro
3. Atribuir um mecanismo de ativação (Barra de ferramentas ou Teclado)
4. Ativar o menu **File > Save As…**
5. Atribuir um nome para o arquivo (com o nome do arquivo atual como default)
6. Selecionar *Rich Text File* no campo *Save as a type*
7. Acionar o botão **Save**

Figura 4: Descrição da tarefa de criação de uma macro para salvar um arquivo no formato *RTF* [DE SOUZA '01].

Este exemplo deixa clara a conseqüência indesejável da replicação de itens léxicos (especificamente o nome do arquivo EXEMPLO.RTF) empregados pelos usuários no momento da gravação de sua macro. O que está ocorrendo é que o paradigma de programação imitativa empregado nos mecanismos de gravação de macro leva em conta a ordem de execução das sub-tarefas, mas é uma programação imitativa liberal. Os gravadores de macros tampouco conhecem o conceito de *type* na sua linguagem, isto é, eles "não possuem variáveis" e, portanto, não têm como realizar generalizações sobre os trechos de processamento gravados. Deve-se fazer uma ressalva, já

[20] *Continuations*.

aludida anteriormente, sobre o que acontece quando se executa uma macro: uma parte arbitrária do contexto no qual a macro está sendo realizada no momento de sua ativação é capturada como parâmetro de entrada. E suas saídas são repassadas para o ambiente da aplicação. Desta forma, o designer arbitrariamente escolhe os elementos que ele quer cristalizar no tempo e os que ele não quer. Arbitrariedades deste tipo podem impactar negativamente a usabilidade deste mecanismo.

Este problema se agrava na medida em que os gravadores de macros não permitem escolha nos caminhos seguidos por um processo gravado — isto é, eles "não possibilitam a especificação de condicionais". E apesar de algumas implementações permitirem iterações sobre conjuntos, por meio de operações predefinidas, nenhuma delas possibilita a especificação direta de iterações controladas por contagem e/ou por condições. Isto é um efeito direto do fato de os mecanismos de gravação de macro não aceitarem variáveis internas aos processos gravados. Como conseqüência destes fatos eles também "não possibilitam uma especificação adequada de iterações".

Um outro grave problema do ponto de vista de compreensão para os usuários finais ocorre quando um usuário quer desfazer[21] a tarefa realizada por uma macro, isto é, ele quer retroceder no tempo. Neste caso, os usuários são pegos de surpresa, pois a maioria dos mecanismos de gravação de macros atuais desfazem somente a última ação realizada pela macro gravada, e não o conjunto total de ações como seria de se esperar. Desta forma, os gravadores de macro "não apresentam um conceito consistente de abstração" [DA SILVA '96], pois não atuam como os outros elementos da interface. Assim o usuário poderá ficar confuso quanto a como interpretar a ação realizada pela macro.

A grande vantagem deste mecanismo é a familiaridade do usuário com a linguagem usada na especificação da extensão, visto que somente são empregadas ações da linguagem de interface da aplicação. Por conseguinte, não é necessário ao usuário aprender uma nova linguagem.

Existem vários tipos de mecanismos de gravação de macros, alguns são específicos para uma aplicação e outros são genéricos para todo o sistema. Deste modo, a usabilidade destes mecanismos estará relacionada ao grau de especificação de ações que sua implementação permite e também à sua capacidade de apresentar a macro como uma abstração igual àquelas definidas pelo designer na interface da aplicação. Assim sendo, apesar de esse mecanismo ser, certamente, o mais empregado nas aplicações comerciais que apresentam alguma forma de extensibilidade, ele também

---

[21] *Undo.*

não tem propriedades suficientes para possibilitar a um usuário criar indiscriminadamente todas as extensões desejadas.

A programação por demonstração

Este mecanismo, algumas vezes chamado de "programação por exemplos", é uma elaboração sobre a idéia de gravação de macros que procura resolver, em parte, os problemas anteriormente citados através da criação de "programas generalizados" a partir de um conjunto de exemplos de tarefas realizadas pelo usuário [CYPHER '93A] [LIEBERMAN '00].

A generalização de exemplos empregada neste mecanismo faz uso de técnicas de Inteligência Artificial (IA), normalmente agentes de software[22] que empregam conhecimento específico do domínio em questão, para resolver os problemas do mecanismo de gravação de macros. Desta forma, o software irá usar, por exemplo, heurísticas para abstrair *types* (variáveis), dos *tokens* da linguagem de interface manipulados durante a criação da extensão. A abstração de condicionais pode ser alcançada através da comparação das semelhanças e diferenças existentes entre os dados de entrada e saída em um conjunto de exemplos, apresentados pelo usuário à aplicação. E a abstração de iterações emprega técnicas de indução sobre o conjunto de exemplos fornecidos, para determinar a intenção do usuário de repetir uma ação.

A grande vantagem deste mecanismo continua residindo na familiaridade do usuário com a linguagem de extensão, pois ele também emprega ações da linguagem de interface como sua linguagem de extensão. O que faz a programação por demonstração melhor que os mecanismos anteriores é o aumento no poder de expressão dado ao usuário através da possibilidade de especificação e uso de *types* na linguagem. Tais elementos colaboram para a alfabetização computacional do usuário, permitindo a ele assimilar conhecimentos básicos de programação. Contudo, os agentes de software geralmente empregados neste mecanismo fazem com que, a princípio, o usuário não precise necessariamente aprender conceitos mais complexos de programação (como, por exemplo, o conceito de iteração), pois eles se encarregam de inferir tais elementos para ele.

Contudo, a detecção de condicionais neste mecanismo é sempre muito complexa, pois, na maioria das vezes, o usuário não fornece todos os casos durante a criação de uma extensão. Na

---

[22] Neste trabalho, sempre que usarmos o termo agente de software estaremos nos referindo a **agentes inteligentes de software**, ou seja, a um software que faz uso de técnicas de inteligência artificial em sua operação.

verdade, ele pode não saber *a priori* quais são todos estes casos. Para superar este problema, algumas implementações permitem que novos exemplos sejam adicionados após a criação da extensão, mas isto não resolve o problema da completude da especificação. Apesar de este mecanismo permitir a especificação dos três tipos de iteração já citados, a detecção de iterações pode acarretar problemas, pois inferir as intenções dos usuários através de técnicas de indução não é uma tarefa precisa. Isto normalmente ocorre devido ao fato de os usuários não serem consistentes em suas ações e não especificarem as tarefas por completo. Além disto, conforme cita Cypher *et al.* [CYPHER '93B], é difícil usar exemplos negativos para a realização de inferências neste mecanismo, pois é complexo demonstrar que um objeto foi selecionado por não apresentar uma determinada característica como, por exemplo, em uma aplicação que usa cores, o fato de ele não ser vermelho. Assim, sempre será mais difícil generalizar sobre os elementos não especificados nos exemplos, uma vez que seu número normalmente é muito grande. Por outro lado, é possível usar uma generalização detectada por um agente de software que foi rejeitada pelo usuário como um exemplo negativo.

O problema mais sério deste mecanismo não é técnico, mas prático. Como ele requer o emprego de agentes inteligentes de software que atuam em background para inferir a necessidade do uso de condicionais, iterações e variáveis no texto da extensão, o usuário terá que inferir sozinho quais as regras que estes agentes estão empregando. Assim, a dificuldade reside em aprender a especificar um conjunto de exemplos (suas etapas e contexto) que seja satisfatório para que o sistema possa inferir corretamente a sua intenção. Deste modo, o problema, que no paradigma de programação paramétrica era o de compreender o significado de um conjunto grande de alternativas léxicas, passa a ser o de prever o significado do conjunto de regras gramaticais que o software emprega na criação de suas extensões. Desta forma, o usuário terá de inferir, através dos resultados apresentados pelo software, qual gramática está sendo usada.

Assim sendo, a usabilidade deste mecanismo está ligada à sua capacidade de interpretar as intenções do usuário inferidas pelas técnicas de IA empregadas na sua implementação, e também à sua capacidade de evidenciar a sua forma de atuação sobre cada exemplo fornecido de forma que o usuário possa facilmente inferir como gerar exemplos satisfatórios.

2.3. O paradigma de programação descritiva

Neste paradigma de programação, a tarefa de *EUP* pode ser comparada à "construção de um plano de ação". Neste caso, o plano será composto pelas sub-tarefas que o usuário quer realizar

na aplicação. Assim, o designer de software deverá disponibilizar, na arquitetura da aplicação, mecanismos que dêem suporte à criação e interpretação destes planos. Estes mecanismos são, normalmente, um ambiente de programação e um interpretador para uma linguagem de extensão. Em vista disso, este paradigma de programação é o que disponibiliza a maior gama de possibilidades de personalização para o usuário final, já que, na maioria das vezes, ele terá o poder de uma linguagem de programação completa a seu dispor.

Assim como no paradigma de programação imitativa, uma extensão neste paradigma registrará a ordem das sub-tarefas do plano de ação do usuário tendo, portanto, uma semântica procedimental. No entanto, é muito comum neste paradigma de programação que uma extensão também apresente partes declarativas, por meio da possibilidade de criação de variáveis e novos tipos nas linguagens de extensão. Este paradigma, como o de programação imitativa, também atua no nível sintático.

Todavia, nos mecanismos que seguem este paradigma, o usuário via de regra não terá ajuda do software na criação de sua extensão, o que exigirá dele conhecimento de conceitos de software e hardware, dos quais ele normalmente não dispõe. Para amenizar este problema Nardi sugere que as linguagens de extensão empregadas pelos mecanismos que seguem este paradigma de programação sejam dependentes do domínio [NARDI '93]. Porém, este dificilmente é o caso, visto que a maioria das linguagens de extensão comercialmente disponíveis atualmente derivam de linguagens de programação empregadas na produção de aplicações completas.

Além deste problema, existe o fato de que o ambiente de programação (isto é, extensão) é, geralmente, desconexo do ambiente de uso normal da aplicação. Assim, quando o usuário resolve criar uma extensão, ele entrará em um modo de operação que emprega um modelo de usabilidade próprio e diferente daquele ao qual ele está acostumado. Aliado a estes problemas, geralmente, está o fato de que o mapeamento entre os elementos da linguagem de interface e os da linguagem de extensão nem sempre apresenta uma relação facilmente dedutível a partir de uma inspeção direta destas duas linguagens. Isto amplia em muito o grau de dificuldade de aprendizado da linguagem de extensão e, portanto, a dificuldade de criação de uma extensão.

Como as linguagens empregadas neste paradigma de programação impõem poucos limites às formas de expressão do usuário, este é o paradigma mais poderoso dentre os discutidos nesta seção. Contudo, este mesmo fato possibilita ao usuário distorcer, ou até destruir, a mensagem original do designer através da adição de funcionalidades totalmente novas à aplicação gerando,

assim, uma nova aplicação e não uma extensão à aplicação original. Este problema é agravado quando os usuários têm que definir diálogos de entrada e saída de informação para as suas extensões. Neste caso, o usuário poderá fazer o emprego de estilos de interação incompatíveis com a estrutura previamente definida na aplicação, conforme será descrito nesta seção no tópico que discute os mecanismos de roteiros. Isto pode causar problemas de compreensão do modelo de usabilidade da aplicação, no que diz respeito à parte estendida.

Desta forma, a usabilidade dos mecanismos que seguem este paradigma de programação, assim como na programação imitativa, também é limitada pela facilidade de aquisição da gramática de sua linguagem de extensão. Contudo, a relação entre os elementos da linguagem de interface e de extensão exerce uma força muito grande sobre esta aprendizagem. Além disso, o suporte dado aos usuários para que estes possam aprender a usar o ambiente de programação e de edição de diálogos de interação também tem influência significativa sobre a usabilidade final dos mecanismos.

Dentre os diversos mecanismos que seguem este paradigma de programação temos os editores de roteiros e de macros, que são amplamente utilizados comercialmente, e a programação visual, que também é empregada comercialmente, porém em menor escala. Além destes mecanismos, podemos citar também os mecanismos com suporte à aprendizagem por desvelamento[23] [DI GIANO '95] e os mecanismos com suporte a analogias [BARBOSA '99] [REPENNING '00B], propostas que, por serem mais recentes, restringem-se ainda à academia.

A aprendizagem por desvelamento

Este mecanismo emprega uma abordagem mista em relação às linguagens usadas na construção de extensões. Ele procura integrar a linguagem de manipulação direta da interface com uma linguagem de extensão textual centrada no domínio da aplicação e, por meio de técnicas de desvelamento progressivo[24] [DI GIANO '95], apresentar ao usuário a relação entre elas. Assim, ele se propõe basicamente a resolver o problema de aprendizagem que aflige a programação por demonstração.

Nas técnicas de aprendizagem progressiva, a linguagem textual empregada é contextualizada no domínio da aplicação [EISENBERG '94], permitindo a especificação de uma tarefa do usuário em mais de um nível. Em vista disso, uma ação na linguagem de manipulação direta da interface poderá

---

[23] *Unfolding.*

[24] *Progressive disclosure.*

apresentar mais de uma expressão lingüística na forma textual. O software, então, empregará o mecanismo de desvelamento, no qual é possível ao usuário questionar o sistema de forma que este lhe apresente diferentes níveis desta hierarquia de expressão, para revelar aos usuários os detalhes e complexidades do sistema de forma sincronizada com o seu nível de conhecimento.

Assim sendo, este mecanismo tenta capitalizar sobre as dificuldades encontradas nas implementações de programação por demonstração fornecendo ao usuário uma forma de ele aprender mais suavemente as regras gramaticais necessárias à construção de extensões no software.

Os roteiros

O mecanismo de roteiro fornece ao usuário uma linguagem de roteiro, que é uma linguagem de programação simples e pequena construída a fim de servir de elemento de ligação entre pequenos programas, ou partes independentes de programas, visando formar uma nova funcionalidade na aplicação. Estas linguagens, geralmente, têm um vocabulário especialmente personalizado para os objetos e ações de um domínio de aplicação.

As linguagens de roteiro suportam variáveis, condicionais e iterações e, portanto, não têm os mesmos problemas enfrentados pelos mecanismos de configuração e de gravação de macros. Infelizmente, as linguagens de roteiro empregadas nas aplicações atuais derivam diretamente das linguagens de programação usadas hoje em dia para o desenvolvimento de aplicações completas — veja-se o *MS VBA*™ empregado no pacote *MS Office*™. A diferença entre elas e as linguagens de programação é que as linguagens de roteiro são interpretadas, permitindo assim que seu usuário tenha uma resposta imediata sobre a ação realizada.

Normalmente associado a estas linguagens existe um ambiente de programação que dá suporte à construção de roteiros. Infelizmente, na maioria das vezes este ambiente é totalmente desconexo do ambiente de trabalho normal do usuário, o que acarreta um grande peso sobre sua usabilidade. Estes ambientes geralmente apresentam um bom suporte, do ponto de vista de um programador, para a edição de extensões. Eles incorporam também ferramentas para permitir a ligação dos roteiros com os elementos de interface, necessários à construção de diálogos para a interação com a extensão que está sendo criada. No entanto, esta ligação é realizada de uma forma totalmente desvinculada do design original da aplicação, podendo o usuário construir diálogos totalmente em desacordo com o estilo de interação preconizado pelo designer de software. Para exemplificar melhor este problema vamos examinar um exemplo, apresentado em [DE SOUZA '01],

retirado do *MS Word*™ que se refere à possibilidade de os usuários criarem formas customizadas para permitir a edição interativa. Neste caso, um conjunto de *widgets* pré-definidos e/ou componentes *OCX* customizados podem ser utilizados para atingir o efeito interativo desejado. Este é de fato um caso avançado de *EUP*, mas assinala alguns pontos que levantamos para estabelecer os limites entre *EUP* e a extensão de aplicações, de um lado, e a re-programação e a atualização delas, do outro.

Os componentes *OCX* são criados fora do *MS Word*™ e do *MS VBA*™ por uma linguagem de programação completa. Bibliotecas de *OCX* podem estar disponíveis em ambientes *EUP* permitindo, assim, que os usuários incluam um ou mais destes componentes em um diálogo de interface. Vamos supor que para uma tarefa de edição repetitiva um usuário decida particularizar a função original de busca do *MS Word*™, de tal forma que agora ela busque um estilo de texto indicado e retorne às páginas onde os estilos ocorrem. As páginas serão mostradas iconicamente em uma janela específica, e o usuário poderá escolher imprimir e/ou ver uma, algumas, ou todas elas. A idéia é apoiar a tarefa de verificação de que o estilo selecionado é impresso de forma adequada por toda a prova do documento.

Vamos supor que, de uma biblioteca de *OCX*, o usuário possa selecionar um componente como o apresentado na Figura 5. Neste componente, o usuário pode abrir e fechar arquivos, procurar um estilo específico, e escolher imprimir ou ver as páginas selecionadas para verificação da qualidade das provas. O *canvas* onde as páginas são mostradas funciona exatamente como a visão de ordenação de páginas nos editores de apresentação de slides. Ele permite ao usuário não somente selecionar um número de páginas, mas também trocar a ordem das páginas (pela manipulação direta dos objetos gráficos), duplicar, apagar e editar as páginas selecionadas.

O problema todo surge devido ao fato de que o componente herda de sua fonte (os editores de apresentação de slides) algumas funções que simplesmente não fazem sentido em um ambiente típico de edição de texto. Por exemplo, a manipulação direta de páginas (que resulta na mudança de sua ordem no documento) não é uma característica desejável para a situação que nós especificamos. Como resultado, ele permite interações que podem ser totalmente indesejáveis no escopo da tarefa que o usuário está tentando realizar como, por exemplo, trocar a ordem linear das páginas do documento ou mesmo apagar todas elas. Deste modo, o efeito da introdução deste tipo de componente que é estranho ao design da aplicação original pode ser claramente perturbador e arriscado nas tarefas de edição de documentos.

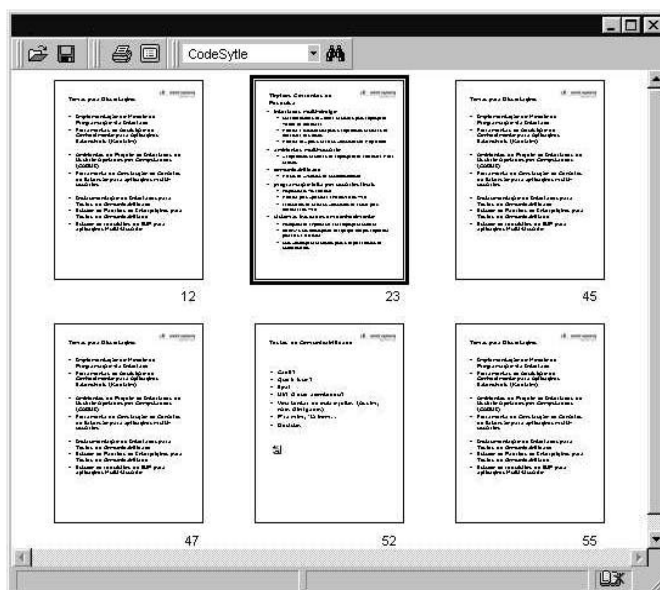

Figura 5: Uso de um componente OCX que não segue os padrões de interação da aplicação.

Os fatos citados até aqui impõem ao usuário final o peso bastante grande de ter de aprender uma linguagem totalmente nova, a qual requer vasto conhecimento de informática, e de ter de aprender a interagir com um ambiente novo para a criação de sua extensão. Ambas as tarefas são bastante complexas e de difícil assimilação e, principalmente, estão fora do escopo da tarefa original do usuário. Logo, podemos ver que este mecanismo não tem grau de usabilidade suficiente para permitir a um usuário final criar suas próprias extensões à aplicação.

A edição de macros

Neste mecanismo, o usuário passará a ter disponível o acesso direto à linguagem de extensão em que as macros são armazenadas no software. Em função disso, ele terá agora a possibilidade de especificar variáveis, condicionais e iterações. O mecanismo de edição de macros é complementar ao de gravação de macros e, quase sempre, eles estão disponíveis em conjunto nas aplicações que os implementam. Esta união normalmente é realizada para disponibilizar ao usuário uma forma de superar as deficiências dos mecanismos individuais.

O uso deste mecanismo pelo usuário poderá ser feito de forma direta ou indireta. Na primeira, o usuário poderá criar sua extensão escrevendo-a diretamente na linguagem de macro. Na segunda, o usuário, após ter criado sua macro por meio do uso de um mecanismo de gravação de macros, terá condições de modificar a estrutura desta macro de forma a torná-la adequada às suas reais intenções. Desta forma, o usuário poderá alterar as construções que tenham sido capturadas

de forma errônea pelos mecanismos de gravação de macros. Em virtude disso, o usuário não estará mais limitado a criar uma réplica do histórico de suas ações.

Os problemas com este mecanismo são similares aos do mecanismo de roteiro. Normalmente, o ambiente de edição da macro opera de forma diferente do ambiente usual de trabalho do usuário e isto, conforme já foi dito, é uma carga a mais contra a sua usabilidade. Além disso, quando o usuário tem que construir elementos de interação para a sua extensão, ele também terá que descrever estes elementos na linguagem de macros. Isto é bastante contra-intuitivo para o usuário, visto que ele não está acostumado a descrever elementos pictóricos na forma escrita. Para aliviar este problema, os ambientes de edição geralmente disponibilizam de um editor de diálogos. No entanto, ao final eles sempre traduzem os elementos pictóricos para a linguagem escrita. Este fato poderá criar problemas para o usuário quando ele tiver que corrigir uma macro defeituosa, uma vez que o texto da macro estará misturado ao dos elementos de interação.

Para melhor exemplificarmos este problema, vamos supor que um usuário necessite de um diálogo como o representado na Figura 6 em uma extensão que ele esteja criando. A maioria dos mecanismos atuais de edição de macros disponibilizarão um editor de diálogos, como o descrito na figura, mas ao final eles exigem que o usuário insira um texto semelhante ao da Figura 7 no texto da extensão para que o diálogo possa ser reconhecido.

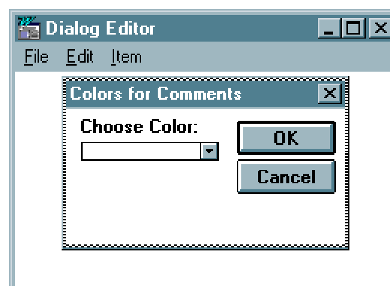

Figura 6: Um exemplo de diálogo criado no *Dialog Editor* do *MS VBA*™.

Deste modo, apesar de este mecanismo apresentar um alto grau de adaptabilidade, por disponibilizar ao usuário uma forma poderosa de geração de extensões, ele peca quanto à sua usabilidade, por exigir do usuário mais conhecimento do que ele normalmente dispõe.

```
Begin Dialog UserDialog 248, 86, "Colors for Comments"
 OKButton 152, 7, 88, 21
 Text 12, 5, 107, 13, "Choose Color:", .Text2
 DropListBox 13, 20, 123, 108, DropListBoxColor$(), .DropListBoxColor
 CancelButton 152, 31, 88, 21
End Dialog
```

Figura 7: Texto gerado pelo *MS VBA*™ equivalente ao diálogo da Figura 6.

A programação Visual

As linguagens de programação visual [CHANG '90] normalmente tentam reduzir a complexidade da criação de extensões por meio da disponibilização de representações visuais para os construtos presentes em uma linguagem de programação convencional. Deste modo, elas buscam capitalizar sobre a facilidade que os seres humanos apresentam de processamento de informação visual. Em geral, elas apresentam todos os construtos presentes em uma linguagem convencional de programação, variando na sua forma de apresentação visual.

Podemos encontrar basicamente duas classes de linguagens visuais nas quais o ambiente e a linguagem visual são: 1) específicos para um domínio e 2) genéricos. As primeiras normalmente aproveitam-se de alguma metáfora* forte para a criação dos elementos de representação para os construtos da linguagem como, por exemplo, o *LabView*™. Já as segundas criam representações abstratas para os construtos encontrados nas linguagens de programação convencionais ou para um nível mais alto de abstração. Em ambos os casos, o grau de usabilidade atingido varia muito e será altamente dependente da escolha da representação e dos níveis de abstração disponibilizados. Green e Petre [GREEN '92] nos mostram que, a despeito das alegadas facilidades de processamento visual atribuídas ao ser humano, nem sempre as linguagem visuais são superiores às textuais nas tarefas de leitura e compreensão de um programa. Rader *et al* [RADER '98] nos mostram também que, mesmo em um ambiente visual com uma linguagem visual altamente elaborada, é necessário o uso de estruturas lingüísticas textuais para ampliar a usabilidade final da linguagem de programação dos usuários finais.

Nos capítulos que seguem, será apresentado o Modelo Semiótico para a tarefa de *EUP*. Este modelo propõe algumas soluções aos problemas presentes nos modelos até agora discutidos. Ele emprega a Engenharia Semiótica e um Modelo de Comunicação para construir um arcabouço teórico que sustentará a definição de uma nova arquitetura de software para as aplicações extensíveis, assim como a definição de um processo de criação de extensões (totalmente ausente nos modelo aqui discutidos) e de uma linguagem-tipo para *EUP*, interligada ao processo proposto.

Capítulo 3

UM MODELO SEMIÓTICO PARA A TAREFA DE *EUP*

Neste capítulo, apresentamos um modelo para a tarefa de programação por usuário final baseado na Engenharia Semiótica e no Modelo da Comunicação Verbal de Jakobson. Partindo da visão proposta pela Engenharia Semiótica de que o software é uma mensagem, sugerimos que a tarefa de uso do software pode ser vista como um processo comunicativo. Assim sendo, empregamos, analiticamente, o modelo de comunicação de Jakobson sobre este processo visando elucidar os componentes nele envolvidos. Da análise do papel de cada componente presente neste processo e da influência que as funções da linguagem, geradas pelo enfoque da mensagem sobre cada um deles, exercem na Interação Humano-Computador, levantamos os requisitos necessários à tarefa de *EUP*. Em seguida, apresentamos os princípios da "Abstração Interpretativa" e do "Contínuo Semiótico" que, segundo nossa abordagem, devem reger a organização dos códigos — as linguagens — atuantes no software extensível para que este seja de boa qualidade. Com estes elementos definidos, apresentamos nosso Modelo Semiótico para a tarefa de *EUP*. É interessante observar que o Modelo de Comunicação Verbal de Jakobson também será empregado, sinteticamente, no Capítulo 6, para apoiar o processo

de realização de extensões pelos usuários finais. A função da linguagem que cada elemento exerce sobre a mensagem será usada de forma construtiva para auxiliar na escolha do código que melhor expresse a intenção comunicativa do designer na interação do usuário com o ambiente de extensão, ampliando, assim, a sua potencial comunicabilidade e usabilidade.

Em nosso trabalho na área de Interação Humano-Computador, empregamos como referencial teórico a abordagem da Teoria Semiótica [ECO'76] [PEIRCE '31], mais especificamente a Engenharia Semiótica [DE SOUZA '01] [DE SOUZA '93]. Na Engenharia Semiótica, o software é visto como uma **mensagem única e unidirecional de alto-nível** do designer de software para o usuário. Em vista disso, sua interface — seu elemento de interação com o usuário — será considerada um **artefato de meta-comunicação** (uma mensagem constituída de forma intencional e regrada que pode gerar ou receber outras mensagens). Deste ponto de vista, a mensagem original que o designer envia para o usuário — a aplicação por ele desenvolvida — apresentará o seguinte significado [DE SOUZA '96A]:

> *"Esta é a minha compreensão de quem você é, do que eu acho você quer e prefere, dos objetivos que você pode atingir com esta aplicação, e das mensagens que você pode e deve trocar com esta aplicação para realizá-los".*

Assim sendo, a atividade central de um usuário no uso do software será a de comunicar-se com a aplicação com o propósito de obter, de forma eficiente e eficaz, as informações de que ele necessita para a execução de suas tarefas. Logo, esta abordagem torna central o **aspecto comunicativo da interface do software** chamando-nos a analisá-lo mais cuidadosamente como um fenômeno de comunicação.

1. O uso de software extensível como um processo comunicativo

Partindo da definição da Engenharia Semiótica que considera o software como um artefato de meta-comunicação, podemos ver o "uso do software" como um processo comunicativo entre o usuário e o preposto do designer — o próprio software. Assim sendo, num cenário típico de uso de software extensível, haverá duas situações bastante distintas a serem consideradas:

- Seu uso na realização de uma tarefa do usuário; e
- Seu uso na criação de uma extensão "a ele mesmo" pelo usuário.

Para analisarmos a inter-relação entre os elementos participantes deste fenômeno de comunicação, tomaremos como base teórica, além da Engenharia Semiótica, o Modelo de Comunicação Verbal de Jakobson [JAKOBSON '60]. Para Jakobson uma **mensagem** é transmitida de um emissor para um receptor. Mas, para que ela seja eficaz, é necessário que seja definido um **contexto** que seja apreensível pelo receptor ao qual ela se refere. Além disto, é necessário um **código** total ou parcialmente comum a ambos e um **canal** — isto é, um meio físico e uma conexão psicológica — que permita a eles entrarem e permanecerem em contato.

Aplicando este modelo a ambas as situações de uso de software extensível, identificamos que o canal de comunicação será sempre o hardware que executa o software e seus periféricos. Este é um canal rico que constitui um meio multimodal o qual possibilita o emprego de múltiplos códigos na emissão e recepção de mensagens. Os demais elementos irão variar nas duas situações que serão analisadas separadamente a seguir.

1.1. O uso normal do software extensível

No primeiro caso do uso de software extensível, o emissor será o designer, uma vez que é ele quem cria o software — a mensagem original. O receptor será ora o usuário (sempre que ele interpreta as mensagens enviadas pelo preposto do designer), ora o próprio software (quando este tem de interpretar as mensagens enviadas pelo usuário por meio de comandos através de sua linguagem de interface — *UIL*[25]). O contexto será definido pelo domínio para o qual a aplicação é construída e pelo sistema computacional[26] no qual ele executará. O código empregado na criação das mensagens que serão trocadas entre a interface do software e o usuário será a *UIL* do software. É interessante observar que, de acordo com a abordagem Semiótica, o background do usuário também influenciará na interpretação das mensagens emitidas pela interface do software fazendo, portanto, parte do contexto. A Figura 8 mostra a instanciação parcial do modelo de comunicação de Jakobson para este caso.

A primeira observação a ser feita sobre este fenômeno é que este é um processo de comunicação em que o seu emissor — o designer do software — não está mais presente diretamente ao processo comunicativo e, portanto, estamos lidando com uma mensagem completa e cristalizada (isto é, uma mensagem na qual tudo o que era para ser dito pelo seu emissor já foi

---

[25] *User Interface Language.*

[26] O conjunto composto pelo sistema operacional e os elementos do seu sistema de interface padrão.

dito), conforme descrito no Capítulo 2. Esta ausência do emissor criará uma lacuna no processo comunicativo podendo gerar problemas à sua efetivação. Tais problemas poderão ocorrer em virtude do fato de que, *a priori*, o receptor ficará impossibilitado de fazer uso completo da função metalingüística (ver definição a seguir), ficando limitado à leitura do help do software.

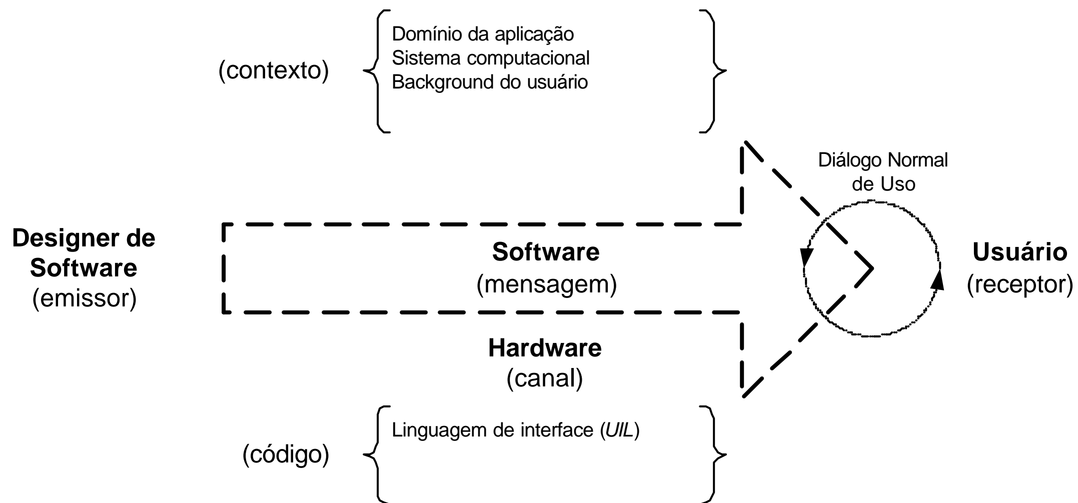

Figura 8: Modelo de Comunicação Verbal de Jakobson parcialmente instanciado para o caso de uso normal do software extensível.

O mapeamento da Figura 8 descreve o uso de software extensível como um processo de comunicação identificando e organizando seus componentes, mas sozinho não nos traz grandes insights. Contudo, Jakobson também descreve as funções que cada um dos componentes de seu modelo exerce sobre a linguagem [JAKOBSON '60]. Estas funções são caracterizadas pelo enfoque dado a um dos componentes do modelo na mensagem e podem ser usadas tanto analiticamente, para ganhar insights no processo de avaliação da comunicabilidade da Interação Humano-Computador [PRATES '00], quanto sinteticamente, para auxiliar na escolha dos elementos do código que melhor expressam a intenção comunicativa do designer durante a construção da mensagem. Assim teremos a função: referencial* (ou denotativa) na qual o enfoque é o contexto da comunicação, expressiva* (ou emotiva) cujo enfoque é o emissor, conativa* (ou apelativa) com enfoque no destinatário, metalingüística* em que o enfoque está sobre o código empregado na comunicação, fática* em que o enfoque está sobre o canal de comunicação e poética* em que o enfoque é a própria mensagem. Jakobson acrescenta ainda que, normalmente, estas funções são empregadas em conjunto na elaboração de uma mensagem e que o enfoque que o emissor dá a uma delas é determinado pelo tipo de informação que pretender transmitir ou obter. Desta forma, é interessante analisarmos o quanto o software, como fenômeno de comunicação, permite ao

designer e ao usuário o uso destas funções na realização de suas tarefas como um processo comunicativo.

A **função metalingüística** é a função da linguagem que está direcionada para a necessidade do remetente e/ou emissor verificar se eles estão usando o mesmo código, assim ela será empregada na elucidação de problemas de interpretação do código em que uma mensagem é transmitida. Logo, para um usuário de software, a perda ou limitação desta capacidade implicará a impossibilidade de esclarecer algumas dúvidas sobre a funcionalidade de elementos interativos da *UIL* que não lhe sejam diretamente interpretáveis (em função de seu conhecimento do domínio e/ou de sua alfabetização computacional e/ou da documentação do software).

É essencial observar que, devido à interface do software ser um artefato de meta-comunicação, a lacuna produzida pela ausência do designer a este processo comunicativo será preenchida pela sua criação — o próprio software — que, desta forma, o representará durante o diálogo metalingüístico com o usuário. Além disso, deve-se notar também que a *UIL* do software forma uma "camada de abstração" que oculta do usuário os demais níveis de linguagem necessários à comunicação com o ambiente computacional e o hardware da máquina. Portanto, para os usuários finais, a *UIL* de um software representa o próprio software, refletindo apenas os elementos comunicativos disponibilizados pelo seu designer. Por este motivo, durante o uso normal do software extensível a maioria das dúvidas dos usuários recairão primordialmente sobre os elementos interativos da *UIL*, e a tarefa de esclarecê-las ficará a cargo da linguagem de explicação do usuário (*UEL*)[27] — denominada também de linguagem de documentação e composta pelo help *online*, os *tooltips* e a documentação do software — que deverá conter as explicações necessárias ao usuário. A *UEL* não deve ser vista como um elemento exterior ao software, mas sim como um componente presente em todo software com a função primeira de representar o designer no diálogo metalingüístico. Assim sendo, a Figura 9 apresenta a instanciação completa do modelo de Jakobson para o caso de uso normal de software.

É interessante observar que, em virtude da impossibilidade de inclusão de explicação para todas as dúvidas dos usuários de um software, a perda do uso irrestrito da função metalingüística pelo usuário final será um problema sem solução total impondo, desta forma, um peso bastante grande sobre a qualidade da *UEL* necessária para minimizar esta perda.

---

[27] *User Explanation Language.*

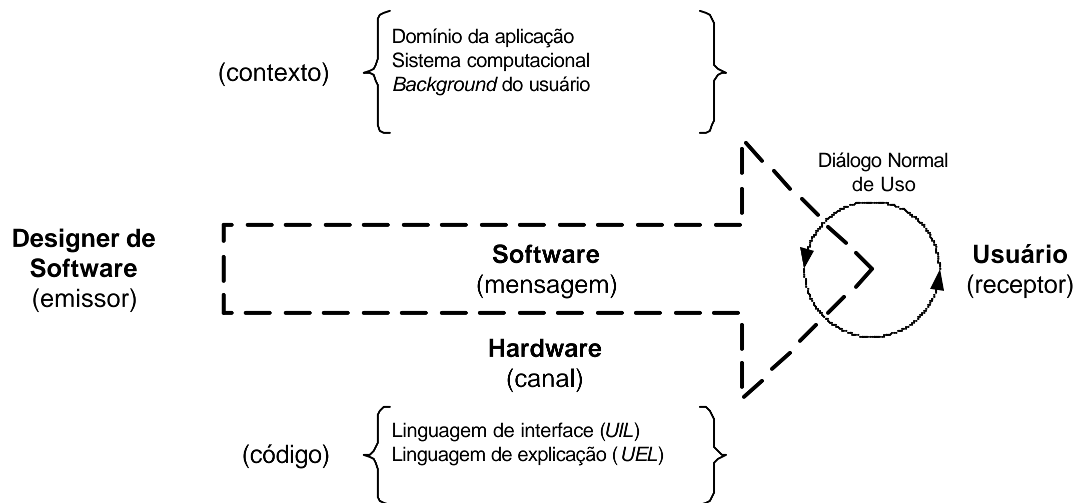

Figura 9: Modelo de comunicação de Jakobson completamente instanciado para o caso de uso normal do software extensível.

Jakobson também comenta que "todo processo de aprendizagem da linguagem (...) faz largo uso de operações metalingüísticas" [JAKOBSON '60], de modo que a impossibilidade de usar parte da função metalingüística afetará também a capacidade de aprendizagem e de aquisição do modelo de usabilidade do software pelo usuário final [LEITE '98]. Este problema será agravado pelo fato de que, segundo a Engenharia Semiótica [DE SOUZA '01], **a *UIL* de um software é uma linguagem única** que se refere exclusivamente aos elementos do modelo semântico único do software em questão. Por exemplo, na realização de uma mesma tarefa em duas aplicações diferentes que atuam sobre o mesmo domínio, o usuário normalmente empregará uma seqüência de interação diferente, que é característica da *UIL* única daquele software. Isto implica que não é possível transferir todo o conhecimento do funcionamento de um software para outro. Assim, toda vez que um usuário estiver utilizando um software novo sua principal atividade será a de **adquirir uma linguagem de interação nova e única**. Esta é uma tarefa de aprendizagem árdua, mas apoiada pelo fato de ser possível transferir de um software para outro os hipocódigos (metáforas) e hipercódigos (estilos) [LEITE '98] empregados na sua interface. Estes elementos contribuem para sensação de *nativeness* do software nos diferentes ambientes computacionais, e para a alfabetização computacional em geral.

A **função referencial** (ou denotativa), é a função da linguagem que enfatiza o referente informando sobre o objeto do discurso. No processo de interação com o software, o usuário estará veiculando e manipulando informação nas mais variadas formas. Para tanto, ele se envolverá em diálogos com o software que fazem referência a, no mínimo, dois contextos diferentes: 1) o

domínio de aplicação do software e 2) o ambiente computacional em que ele executa. Assim, a função referencial exercerá papel primordial para esclarecer a quais objetos o usuário e o software estão se referindo em um determinado momento e em que contexto isto acontece. Em um cenário como este, o usuário fará uso dos periféricos, que compõem o canal de comunicação, para indicar ao software sobre quais elementos operar e o software ressaltará na *UIL* os elementos sobre os quais ele está atuando ou a que precisa que o usuário dê atenção.

Assim sendo, a função referencial exercerá um papel altamente relevante sempre que o software apresentar dois ou mais modos — contextos — de operação diferentes ou sempre que seja imprescindível chamar a atenção do usuário sobre um objeto ou grupo de objetos. No caso do uso de modos de operação diferentes, existe a necessidade de marcar na *UIL* a mudança do contexto de discurso, uma vez que o usuário precisará saber identificar em cada contexto: quais objetos são manipuláveis, qual os comportamentos destes objetos e quais as funcionalidades válidas sobre estes objetos. No caso da operação sobre grupos de objetos, existe necessidade de marcar na *UIL* a distinção entre um objeto simples e um composto de modo que o usuário possa ter as mesmas informações que teria no caso de contextos diferentes. A não marcação do contexto na *UIL* faz com que o usuário possa perder compreensão sobre o modelo de usabilidade da aplicação e, portanto, não conseguir realizar suas tarefas corretamente.

A existência de contextos diferentes de atuação impõe outra necessidade ao software, a de ter uma *UEL* que seja sensível a esta diferença. A maioria das *UELs* atuais são, no máximo, sensíveis aos elementos léxicos da *UIL*, dando respostas idênticas a situações de uso bastante diferentes. Esta não diferenciação no diálogo metalingüístico terá uma grande influência sobre a aquisição do modelo de usabilidade da aplicação pelo usuário, uma vez que ele não terá como distinguir a natureza de suas dúvidas através da diferenciação das situações em que elas ocorrem.

Assim sendo, a despeito de estarmos analisando o uso normal do software extensível, é importante observar que somente quando um usuário compreender suficientemente o modelo de usabilidade do software para saber que a funcionalidade por ele requerida não se encontra disponível, é que ele poderá ter a intenção de criar uma extensão a este software. Isto torna as funções metalingüística e referencial essenciais à tarefa de *EUP* desde a fase inicial de aquisição do modelo de usabilidade do software, em virtude da existência de no mínimo dois contextos diferentes de uso. Portanto, um requisito para qualquer software extensível é que ele:

Disponibilize uma UEL que seja tanto quanto possível sensível ao contexto de operação do usuário de forma a auxiliá-lo adequadamente na aquisição do modelo de usabilidade do software.

A **função conativa** (ou apelativa) é a função da linguagem que evoca uma ação do seu receptor e, normalmente, está relacionada a vocativos e imperativos na linguagem verbal. No processo de interação com o software, ela estará ligada à necessidade de chamar o usuário à ação, quando o software precisa "passar a vez" de atuar no diálogo ao usuário. Em vista disso, o designer tem no correto uso desta função na *UIL* uma forma de indicar, ou até forçar, a necessidade de o usuário realizar determinada seqüência de ações no software para que o diálogo possa continuar. A não sinalização por parte do designer desta necessidade poderá causar ao usuário a sensação de incapacidade por não saber exatamente quando e como deve atuar na realização de uma tarefa. Do ponto de vista do usuário, a função conativa será empregada sempre que ele estiver ativando a realização de uma determinada funcionalidade e, por conseqüência, estiver passando a vez de atuar no diálogo ao software.

A **função fática** é a função da linguagem evidenciada por uma troca profusa de fórmulas ritualizadas cujo único propósito é testar o canal de comunicação. Logo, no processo de interação com o software, ela estará ligada à necessidade de alertar o usuário sobre alguma alteração na condição deste canal durante a execução de uma tarefa. Esta necessidade é comum em situações em que a tarefa requerida pelo usuário apresenta uma resposta lenta (por exemplo, por fazer muito uso da *CPU* ou por ter de acessar um *site* na *Internet* que seja muito lento) e também quando, por qualquer razão, seja necessário bloquear o uso dos dispositivos de entrada de dados (*mouse*, teclado, microfone, etc.). No primeiro caso, o designer deverá empregar uma mensagem enfocando a função fática na *UIL* para informar ao usuário que a resposta para sua ação é lenta e que, apesar de não aparentar, o canal ainda está funcionando. Caso contrário, o usuário não conseguirá saber qual ação tomar em seguida para dar continuidade ao diálogo. No segundo caso, o designer deveria enfocar a função fática na *UIL* para informar ao usuário que o bloqueio do canal de comunicação é temporário e por quanto tempo ele ficará neste estado. Caso contrário, como os dispositivos de entrada de dados são os meios dos quais o usuário dispõe para expressar mensagens com estrutura fática, ele perderá sua capacidade de checar o canal e, portanto, não saberá também como continuar o diálogo.

A **função poética** é a função da linguagem em que o enfoque da mensagem é ela própria, valorizando sua forma de expressão retórica. Do ponto de vista do processo de interação do

usuário com o software, esta função está ligada aos padrões de interação — estilos e metáforas — empregados na *UIL* do software. Como descrito anteriormente, estes padrões são compostos por um conjunto de hipocódigos e hipercódigos que serão responsáveis pela sensação de *nativeness* entre o software que é operado em um mesmo ambiente computacional. É importante observar que, apesar de estar relacionada com a forma da mensagem, esta função também transmite informação. Os estilos e as metáforas da interface do software expressam, de forma indireta, parte das decisões de design tomadas durante a construção do software pelo seu designer. Estas decisões refletem-se na aparência de um software dentro do seu ambiente computacional e, portanto, podem facilitar em muito a interpretação dos elementos encontrados na *UIL*. Isto explica a razão pela qual o emprego de elementos de interação que não sigam os estilos padronizados no ambiente computacional irá requerer do usuário um esforço extra de interpretação, tornando mais difícil a assimilação do modelo de usabilidade do software e dificultando assim o seu uso.

A **função expressiva** (ou emotiva) é a função da linguagem que visa à expressão direta da atitude de quem fala em relação àquilo de que está falando. No processo de interação do usuário com o software, esta função está ligada à expressão do designer (representado pelo seu preposto) em relação à sua percepção do estado do receptor, suas crenças e suas atitudes frente à sua interação com a aplicação. Como o designer não está diretamente presente neste processo de comunicação e o seu representante — o software — apresenta *a priori*[28] inteligência limitada, o uso desta função torna-se limitado à interpretação das formas de expressão emotivas permitidas no código artificial que é a *UIL*. Todavia, esta função terá utilidade quando for necessário informar ao usuário, por exemplo, que a "vez" de atuar no diálogo é do software e que é inútil insistir em alterar esta ordem porque não é possível interromper a operação em questão. Outro emprego desta função será visto quando o software precisar reportar erros ao usuário, caso em que o designer deverá enfatizar na *UIL* a função emotiva para transmitir ao usuário a impossibilidade de continuar a tarefa em questão, em virtude da ocorrência "lamentável" de erros na sua execução.

Neste ponto concluímos a análise da primeira situação de uso do software extensível. Desta análise pudemos detectar, por meio da projeção do Modelo de Comunicação Verbal de Jakobson sobre o processo de uso do software, quais os elementos centrais neste processo. Ainda por meio desta projeção, identificamos a influência exercida pelas funções da linguagem sobre a Interação Humano-Computador quando vista como um processo de comunicação. Concluímos então que,

---

[28] Existem trabalhos que têm a intenção de inserir capacidade emocional em interfaces de software. Estas interfaces não estão sendo alvo deste estudo.

do ponto de vista do uso normal do software extensível, o usuário competente em sua utilização será aquele que melhor compreender seu modelo de usabilidade de forma a ser capaz de inferir a seqüência de ações necessária à realização de suas tarefas. Desta forma, mostramos que é essencial a presença, no software extensível, de um mecanismo de explicação para minimizar a perda da função metalingüística pelo usuário final.

1.2. O uso do software extensível na criação de extensões

No segundo caso do uso de software extensível, o software poderá ter suas funcionalidades ampliadas por meio da criação de extensões. Como o designer não mais está presente ao diálogo, estas extensões somente poderão ser realizadas pelo usuário. Logo, neste caso, haverá dois emissores: o designer do software (que cria a mensagem original) e o **usuário como designer** (que criará extensões à mensagem original). Este duplo papel do usuário é um dos elementos críticos no software extensível, pois o usuário não tem conhecimento algum do processo de design de software e, na maioria das vezes, não deseja perder seu tempo para adquiri-lo. Desta forma, o designer deverá prover mecanismos internos ao software que auxiliem o usuário na realização desta tarefa. Deste ponto de vista, a mensagem original que o designer envia ao usuário terá seu significado alterado no software extensível, sendo restringida como segue:

> *Esta é a minha compreensão de quem você é, do que eu acho que você quer e prefere, dos objetivos que você pode atingir com esta aplicação, das mensagens que você pode e deve trocar para realizá-los, e da forma de você estender a capacidade desta aplicação para atingir estes objetivos de forma mais eficiente.*

Logo, a atividade central de um usuário no uso do software extensível continuará a ser a de comunicar-se com a aplicação com o propósito de obter, de forma eficiente e eficaz, as informações de que ele necessita para a execução de suas tarefas. Contudo, agora ele também poderá comunicar a sua intenção de alterar o código — a linguagem — com o qual ele se comunica a fim de tornar os diálogos mais eficientes e eficazes. Este fato caracteriza o uso de software extensível como uma **atividade de comunicação e de representação** em que o usuário poderá expressar o modo pelo qual ele deseja interagir na realização de uma tarefa específica.

Para determinarmos os mecanismos de suporte necessários às atividades do usuário é preciso que analisemos o efeito que o seu duplo papel exerce sobre o processo comunicativo. Para isto, primeiramente, é necessário caracterizar o que será considerada uma extensão ao software.

Como discutido anteriormente, para o usuário a *UIL* é o software, assim podemos concluir que, independentemente da forma de realizar as extensões, elas deverão resultar na "alteração do código da *UIL*", pois é este o código que o usuário usa para se comunicar com o software.

O fato de a *UIL* ser a forma de comunicação do usuário e também o componente que será alterado nos aponta a necessidade de um mecanismo de apoio ao usuário que introduza um novo contexto de operação do software — um contexto de realização de extensões. Este novo contexto será necessário para distinguir o uso normal da *UIL*, no qual o usuário estará se referindo a elementos do domínio e/ou do ambiente computacional, do seu uso metalingüístico, no qual o usuário estará se referenciando a elementos do meta-modelo do software. Uma vez que o usuário precisa referenciar elementos metalingüísticos, também será necessária a introdução de um novo código — uma **linguagem de extensão do software** (*EUPL*)[29] — que lhe possibilite comunicar as operações metalingüísticas requeridas para a criação de suas extensões, pois os códigos existentes não lhe possibilitam esta tarefa. A Figura 10 mostra a instanciação do Modelo de Comunicação Verbal de Jakobson para este caso.

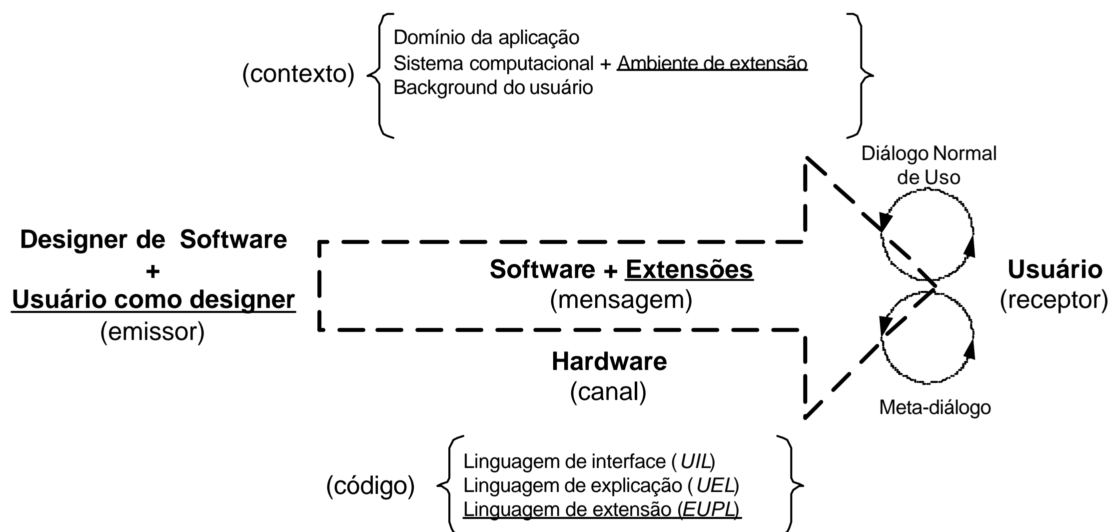

Figura 10: Modelo de Jakobson instanciado para o caso de uso do software extensível na criação de extensões ao próprio software.

A introdução destes dois novos componentes no software extensível resolve apenas parcialmente o problema de suporte ao usuário, pois eles somente fornecem os meios de realizar a tarefa, mas não descrevem o processo para realizá-la. Para estabelecermos um modelo satisfatório

---

[29] *End-User Programming Language.*

para a tarefa de *EUP*, é necessário estudar os reflexos da alteração do código da *UIL* sobre o seu uso como meio de comunicação entre o usuário e o software.

Do ponto de vista do usuário, o maior reflexo da possibilidade de alteração do código da *UIL* é que, ao ocupar o papel de emissor na criação de extensões ao software, a sua competência terá que mudar visto que o foco de seu discurso passará a ser os elementos do meta-modelo do software. Assim sendo, não bastará mais a ele saber expressar-se bem na *UIL*, ele também terá que saber expressar-se de forma produtiva na *EUPL*. Esta exigência decorre do fato de que a realização de extensões exige do usuário que ele "produza descrições lingüisticamente corretas" das ações que ele quer que a extensão realize no software. Isto implica que o usuário terá que saber tanto a sintaxe como a semântica de *EUPL*, que são elementos metalingüísticos em relação a *UIL*.

Nardi [NARDI '93] nos alerta para o fato de que as dificuldades de programação encontradas pelos usuários finais estão relacionadas à "baixa expressividade" das linguagens de extensão a eles disponibilizadas. Como solução para este problema, ela propõe que uma *EUPL* seja uma linguagem de propósito especial e fortemente ligada ao domínio da aplicação. Porém, segundo a Engenharia Semiótica, estas linguagens devem não somente ser de propósito específico mas também **linguagens únicas de extensão** [DE SOUZA '01]. Esta unicidade das *EUPLs* está diretamente relacionada à natureza única das *UILs*, as quais elas irão estender, e à sua dependência com o domínio da aplicação. Devido a esse aspecto de unicidade, as *EUPLs* apresentam um grau de dificuldade extra para seu aprendizado por parte dos usuários finais e, portanto, um requisito para qualquer software extensível é que ele:

Disponibilize mecanismos que auxiliem o usuário na aquisição da gramática e semântica da sua EUPL.

As necessidades metalingüísticas do usuário, porém, vão mais além. A possibilidade de alteração da *UIL* do software faz com que ele esteja exposto a um processo constante de aprendizado do uso deste software, pois a cada extensão serão inseridos novos elementos ao código da *UIL* possibilitando a ele trocar novas mensagens com a interface do software para a realização de suas tarefas. Este fato faz com que o modelo de usabilidade do software "nunca se estabilize" e, portanto, o papel da função metalingüística será ainda mais marcante neste caso de uso do software extensível do que no anterior.

Para o designer, a possibilidade de alteração do código da *UIL* acarretará uma mudança de ponto de vista durante o design do software, pois será necessário alterar a *UEL* não somente para que ela apóie as necessidades metalingüísticas do usuário em relação ao ambiente de extensão e ao uso da *EUPL*, mas também para que ela seja capaz de dar suporte à explicação dos novos elementos inseridos na *UIL* por meio das extensões realizadas pelo usuário. Esta última exigência implica que **a *UEL* terá de ser atualizada a cada nova extensão** inserida no software. Sem esta atualização não será possível manter a função metalingüística atuando de forma satisfatória, pois ela falhará quando for necessário acessar informações sobre estas extensões. Por conseguinte, outro requisito para qualquer software extensível é que ele:

> *Disponibilize uma UEL que seja evolutiva de forma que possa ser adaptada às necessidades de esclarecimentos das dúvidas do usuário decorrentes das extensões à UIL do software criadas por ele.*

É interessante observar que os requisitos até agora levantados vão ao encontro direto da proposta de Adler e Winograd [ADLER '92] refletindo o problema da aquisição de conhecimento do software por parte dos usuários finais para a realização de suas tarefas, sejam elas normais ou de extensão. Porém, nenhum deles garante que as extensões ao software produzidas por estes usuários não venham a invalidar o conhecimento já adquirido. A fim de evitar a perda deste conhecimento é indispensável que as extensões ao software não sejam destrutivas e, portanto, que as extensões geradas pelos usuários somente possam adicionar, reorganizar ou personalizar a *UIL* dentro de uma faixa de tipos possíveis de **modificações antecipadas pelo designer**, de forma a não corromper o seu design original. Devido a esta limitação nos tipos de extensões e ao fato de que estas extensões ocorrerão sobre a *UIL*, a tarefa de realização de extensões em *EUP* será equivalente à **geração de *tokens* de tipos interativos realizados ou potenciais** que deverão ser intrínsecos à concepção da mensagem original do designer para o usuário [DE SOUZA '01]. Esta visão está de acordo com o mecanismo de programação por demonstração [CYPHER '93A], no qual os tipos interativos são usados para inferir extensões pretendidas e somente *tokens* específicos da *UIL* do software que pertençam a estes tipos podem ser gerados pelos mecanismos de *EUP*.

Estas observações procuram deixar claro que **extensão é estritamente diferente de programação**, pois, na primeira, pressupõe-se a existência de algo a ser estendido (o modelo concreto de usabilidade projetado pelo designer original do software), enquanto que, na segunda, pode-se partir diretamente de um modelo conceitual abstrato (resultado de uma análise do domínio,

das classes de usuários a serem atendidas e das tarefas que se pretende apoiar). Isto implica que um usuário no papel de designer não poderá introduzir novos elementos que não tenham relação com o modelo de usabilidade original do software e, portanto, um novo requisito para qualquer software extensível é que ele:

> *Disponibilize mecanismos que possibilitem somente a criação de extensões monotônicas nas quais os usuários serão criadores de tokens interativos dos types pré-definidos pelo designer do software.*

Contudo, não basta garantir a exclusão de elementos estranhos ao modelo de usabilidade do software para evitar a perda de conhecimento com a criação de extensões, é preciso também que as extensões construídas pelo usuário respeitem o **Ciclo Mínimo de Interação*** necessário à criação de diálogos completos na *UIL*. No Modelo Semiótico aqui proposto, este ciclo mínimo é composto por três passos que podem ser descritos como [DE SOUZA '01]:

- O software diz alguma coisa para o usuário — isto é, a mensagem do designer, descrevendo as funcionalidades disponíveis, é passada ao usuário através da *UIL*;
- O usuário diz alguma coisa ao software — isto é, o usuário define a ação que quer executar para realizar parcialmente, ou totalmente, sua tarefa; e
- O software responde ao usuário — isto é, o software apresenta o resultado da efetivação de sua funcionalidade.

É interessante observar que este ciclo diferencia-se do ciclo de interação dos modelos da tarefa de *EUP* analisados no Capítulo 2 devido ao fato de aqueles, geralmente, considerarem somente os dois últimos passos descritos acima. Desta forma, aqueles modelos não levam em conta o fato de que o designer comunica ao usuário, através da interface, a estrutura de funcionamento do software e que uma falha nesta comunicação resultará na sua baixa usabilidade, como nos mostra Prates *et al.* em [PRATES '00].

Assim, para garantir que o ciclo mínimo de interação seja respeitado, é necessário que a *EUPL* tenha uma **noção de texto** que seja sintaticamente distinta, ou uma construção de mais alta ordem que aquela equivalente às instruções, bloco ou programa executável, de tal forma que se possa associar uma **interpretação pragmaticamente*** **válida** às construções de texto. Conseqüentemente, um outro requisito para todo software extensível é que ele:

Disponibilize mecanismos que auxiliem o usuário a distinguir os textos — na EUPL — que são pragmaticamente corretos e válidos na UIL daqueles que não o são, garantindo assim o Ciclo Mínimo de Interação.

A manutenção do Ciclo Mínimo de Interação tem relação direta com a função poética da *UIL* do software visto que, em nossa abordagem, toda extensão provoca uma alteração da *UIL*. Assim sendo, para que a *UIL* seja de fácil compreensão e aprendizagem, será necessário que as extensões criadas pelo usuário respeitem os padrões (estilos e metáforas) empregados no design original do software. Novamente, o problema da falta de conhecimento computacional do usuário fica evidente. Neste caso, o usuário desconhecerá as diretrizes de estilo recomendadas para a criação de diálogos de interface no ambiente computacional e os estilos pessoais empregados pelo designer. Por conseguinte, a fim de manter a continuidade semântica da *UIL* da mensagem original, o designer deverá disponibilizar no ambiente de extensão mecanismos que auxiliem o usuário na construção de diálogos de interação necessários à sua extensão, de modo que estes diálogos respeitem as diretrizes de estilo do ambiente computacional e as regras por ele utilizadas durante a construção da mensagem original.

É fundamental observar que a natureza metalingüística dos objetos manipulados pela *EUPL*, dentro do ambiente de extensão, faz com que a função referencial seja mais importante ainda neste caso do que no uso normal do software extensível. Este fator aumenta quando o ambiente de extensão for a própria *UIL*, como ocorre nos mecanismos de extensão do paradigma de programação imitativa e na programação Via Interface [BARBOSA '99]. Neste caso, a distinção entre o ambiente de uso normal do software e seu ambiente de extensão ficará bastante nebulosa e, por conseguinte, os elementos da *EUPL* se confundirão com os da *UIL*. Assim sendo, caso não haja uma boa marcação na interface do contexto ao qual um elemento está ligado em determinado instante, o usuário tenderá facilmente a confundir as funcionalidades válidas sobre este objeto. Em virtude disso, ele não conseguirá realizar uma extensão correta ou não entenderá o modelo de usabilidade do software. Para complicar ainda mais esta situação, é comum que o designer do software desative alguns elementos da *UIL* (e algumas vezes até mude o modo de uso dos periféricos de entrada de dados) durante o processo de realização da extensão, o que confunde ainda mais o usuário. Como reflexo secundário, estas alterações podem afetar o uso da função fática pelo usuário. Logo, é de suma importância no software extensível que o designer deixe claro ao usuário, por meio da *UEL*, que, ao criar uma extensão, ele estará falando sobre um novo elemento da *UIL* e, portanto, estará realizando uma "ação metalingüística". É essencial dizer ao

usuário que, neste caso, alguns elementos da *UIL* (que estão diretamente ligados à função metalingüística) não operarão da mesma forma.

Outro fato relevante, resultado de observações experimentais realizadas com uma versão inicial da *EUPL* proposta neste trabalho, a ser descrita no Capítulo 5, é que será necessário guiar o usuário na tarefa de realização de sua extensão. Nossos experimentos indicam que os usuários leigos, na sua maioria sem conhecimento de programação, e mesmo os que são programadores, ao criarem extensões, normalmente concentram-se na parte do processamento da funcionalidade da extensão, em detrimento das sub-tarefas de entrada e saída de dados e de tratamento de erros. Este fato resulta em problemas na manutenção do Ciclo Mínimo de Interação e no tratamento de erros, afetando a função expressiva da *UIL* do software. Por conseguinte, é necessário que o designer faça uso extensivo da função conativa durante o processo de realização da extensão para guiar os usuários a completar todas as sub-tarefas necessárias à produção da extensão.

Para garantir que as extensões criadas pelos usuários sejam válidas e corretas, será necessário realizar sua verificação sintática e sua validação semântica. A verificação sintática pode ser parcialmente obtida por construção, podendo sua verificação final ser feita de forma automática pelo software. Porém, a validação semântica requer a intervenção direta do usuário, visto que é ele quem sabe qual é a real função da extensão. Como existe uma possibilidade de o software parar de funcionar durante a fase de teste da extensão em função de um erro na sua implementação, é de suma importância manter o usuário informado sobre a situação atual do canal de comunicação para que ele não se sinta perdido e saiba quando é necessário atuar para continuar o diálogo. Isto implica que a função fática deverá ser ressaltada nesta fase do processo de construção da extensão. Contudo, não é somente neste caso que ela será importante. No software extensível, é essencial que o designer sinalize claramente as possibilidades de interrupção das tarefas que estão em andamento, de forma que o usuário possa sempre abandonar uma ação em que ele se encontre perdido (sem saber o que fazer), ou que ele possa postergar uma ação sem muita importância, ou simplesmente desistir de realizar a ação.

Por último, para suprir a necessidade de explicação para o uso das extensões realizadas na aplicação, será necessário que os usuários finais forneçam uma explicação mínima para cada uma das extensões por eles realizadas. Esta explicação será introduzida na *ADKB*, juntamente com a nova extensão, para suprir a perda da função metalingüística causada pela ausência do designer de software (neste caso, o próprio usuário final).

Neste ponto encerramos a análise dos reflexos diretos que a introdução da possibilidade de estender o código da *UIL* traz para o software extensível. No entanto, existe mais um ponto importante a observar. Como vimos, a introdução da *EUPL* ao conjunto de códigos manipuláveis pelo usuário (possibilitando a alteração da *UIL*) e a necessidade de alteração dinâmica da *UEL* (para suprir as necessidades metalingüísticas do usuário) demonstram que o inter-relacionamento entre os códigos existentes em um software extensível é bem mais complexo que o encontrado em um software não-extensível. Portanto, é essencial que se investigue a natureza e o perfil dos códigos — das linguagens —, que o usuário terá que utilizar para realizar a tarefa de *EUP*, para que se possa determinar a classe de códigos que perfazem um bom ambiente comunicativo nesta situação.

2. A natureza e perfil dos códigos no software extensível

Na seção anterior, discutimos a qualidade do ambiente necessário ao software extensível para auxiliar os usuários finais na tarefa de *EUP*. A despeito de este ser um elemento primordial na qualificação de um software extensível, a principal ferramenta do usuário ainda é o código que ele emprega na interação com o software. Em um software extensível este código estará dividido em dois elementos principais — a *UIL* e a *EUPL* — e um secundário — a *UEL*. Com exceção da *UEL*, que normalmente emprega códigos naturais, os outros dois códigos geralmente são compostos de "linguagens artificialmente geradas". Os códigos artificiais atuais são restritos e comumente não contêm todos os componentes presentes em um código natural escrito, a saber: o morfológico, o léxico, o sintático, o semântico e o pragmático. Mais especificamente, os componentes "morfológico" e "pragmático" não estão presentes na maioria destes códigos. Esta ausência é fator primordial na deficiência de expressividade apresentada por estes códigos resultando na dificuldade de seu uso pelos usuários finais, problema este já apontado por Nardi [NARDI '93].

A ausência do componente morfológico nos códigos artificiais tem vários reflexos sobre o seu poder de expressão. Por exemplo, a frase "CORTE OS TOMATES", que é uma sentença imperativa, contém a especificação de uma iteração implícita através do uso da palavra "TOMATE" e do determinante "O" no plural. Este exemplo nos mostra que a simples alteração de uma palavra, uma operação morfológica, permite uma modificação significativa na estrutura da ação que a sentença que a contém especifica. Devido ao fato de os códigos artificiais não fazerem uso deste recurso lingüístico, eles obrigam seus usuários a especificar um conjunto de comandos complexos para expressar o mesmo tipo de ação. O componente morfológico também é responsável, na

linguagem natural, pela marcação de vários outros aspectos como, por exemplo, do tempo de ocorrência de uma determinada ação. Estes aspectos podem ser de grande valia para facilitar a comunicação do usuário com o software, porém eles também não são levados em consideração nos códigos artificiais atuais.

Peirce [PEIRCE '31] já nos alerta para o fato de os signos serem interpretados somente em relação aos objetos da situação na qual alguém (o usuário) faz uso da linguagem — isto é, o **contexto** de uso da linguagem determina as **regras pragmáticas*** que definirão a **validade de um texto** de um determinado código **em uma dada situação**. A influência do contexto na interação humano-computador está claramente exposta por Suchman em [SUCHMAN '87] e sua ação é preponderante não somente sobre a usabilidade de um software, mas também sobre sua extensibilidade.

Assim, a ausência do componente pragmático nos códigos artificialmente gerados cria uma restrição significativa sobre o poder expressivo destes códigos. Ela exige que seus usuários tenham sempre que explicitar o contexto em que um texto deve ser interpretado e, por conseqüência, tenham que utilizar muito mais código para expressar uma mesma idéia. Por exemplo, se dentro de um contexto de leitura de um livro um professor diz a um aluno: "ENCONTRE TODAS AS PALAVRAS 'PODER' NO TEXTO", ele está querendo dizer o mesmo que: "folheie o livro capítulo por capítulo, parágrafo por parágrafo, sentença por sentença e ao identificar a palavra 'poder' ou sua variante plural 'poderes', marque o local em que ela se encontra". O conhecimento de que o texto é um livro, que é composto de capítulos e que estes são compostos de parágrafos, que por sua vez são compostos por sentenças e estas de palavras, é um conhecimento partilhado que pertence ao contexto de trabalho do aluno e do professor e dirigirá a interpretação da sentença enunciada pelo professor. Sem este conhecimento não seria possível interpretar corretamente esta sentença. A existência deste contexto comum permite ao professor omitir um grande volume de informações que podem ser facilmente inferidas pelo aluno, informações estas referentes à composição estrutural do livro.

Se, por exemplo, um usuário quiser especificar uma extensão que realize a tarefa acima descrita, utilizando-se dos códigos artificiais atualmente disponíveis, ele terá de especificar explicitamente todas as iterações necessárias para percorrer cada uma das relações todo-parte mais a variação de número (singular/plural) existente entre os elementos que compõem o livro, o que é um tarefa árdua. Desta forma, a ausência dos componentes morfológico e pragmático no código

impõe a necessidade de uma relativa "superespecificação das tarefas" pelo usuário, dificultando, assim, a realização de suas extensões. Deste modo, fica claro que a natureza dos códigos atualmente empregados no software extensível faz com que eles sejam inadequados para os usuários finais de tais aplicações, uma vez que eles apresentam um poder expressivo muito aquém do usado por estes usuários na especificação de suas tarefas diárias.

Logo, para reduzir o problema da baixa expressividade levantado por Nardi, é necessário que os códigos empregados no software extensível estejam mais próximos da linguagem de uso diário do usuário final. Por conseguinte, um outro requisito para todo software extensível é que ele:

> *Disponibilize mecanismos que possibilitem aos códigos por ele utilizados expressar significado através de elementos morfológicos e pragmáticos, tais como os usuários finais estão acostumados a fazer em linguagem natural.*

Uma vez discutida a natureza dos códigos que estão presentes em um software (seja ele extensível ou não), é importante também definirmos o perfil do relacionamento destes códigos. Nas subseções que se seguem discutiremos dois princípios que procuram abarcar o relacionamento destes códigos no software comum e no extensível.

2.1. O princípio da Abstração Interpretativa

No modelo do uso de software discutido na Seção 1 deste capítulo assinalamos o fato de que para o usuário final a *UIL* é o software. No entanto, é essencial observar que este fato somente será verdade caso a *UIL* realmente funcione como uma camada de abstração sobre o nível de implementação do software. Isto somente ocorrerá caso o usuário consiga interpretar os códigos empregados na *UIL* recorrendo apenas ao conhecimento disponível no seu contexto de comunicação. Qualquer informação a mais, necessária à interpretação de algum signo da *UIL*, implica que o usuário precisará de conhecimentos que remontam ao nível de implementação do software e, portanto, a barreira de abstração gerada pela *UIL* será quebrada.

Para melhor compreendermos a afirmação acima, apresentamos um exemplo típico de quebra da abstração da *UIL*, gerado por um editor de texto comercial em resposta a uma ação do usuário. Neste caso, o usuário manda gravar um arquivo em um disco flexível, porém, como o disco não se encontra no *driver* ou está bloqueado para escrita, a aplicação devolve a mensagem de erro apresentada na Figura 11. O diálogo apresentado nesta figura emprega o elemento "*Error 70*",

que não faz parte do domínio da aplicação, e que também não consta de parte alguma da *UEL* do software e muito menos faz parte do background computacional de um usuário final típico. A compreensão do significado deste elemento envolve conhecimento que pertence ao domínio do ambiente computacional no qual o software está executando, conhecimento que o usuário não possui.

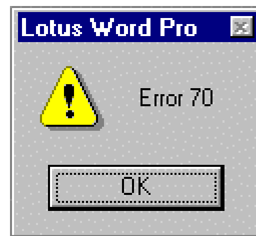

Figura 11: Um exemplo de uma mensagem de erro que emprega um código que pertencem ao nível de implementação do software .

Como podemos ver, este é um problema que poderá ocorrer em qualquer tipo de software e está diretamente ligado à forma de relacionamento entre os códigos interativos de sua interface e os empregados na sua implementação, que definem a semântica operacional da *UIL*. O **princípio da Abstração Interpretativa**, proposto pela Engenharia Semiótica [DE SOUZA '01], procura avaliar quão bem um código artificial abstrai outro sobre o qual ele está implementado, do ponto de vista de sua interpretação. No caso do software extensível, ele procura avaliar, do ponto de vista de interpretação, quão bem a *UIL* abstrai a faixa de funcionalidades predefinidas e estendidas do software. Deste modo, este princípio serve como balizador da qualidade dos códigos empregados no software (seja ele extensível ou não).

Deste modo, dizer que uma linguagem computacional é uma Abstração Interpretativa de uma ou mais outras linguagens quer dizer que um usuário deve ser capaz de compreender e interagir com a primeira, sem conhecimento algum das especificações expressas nas demais linguagens. No caso de software extensível típico, a *UIL* é composta de uma parte fixa e uma parte extensível (*UILx*). Assim, depois que uma ou mais extensões forem criadas, a *UIL* estendida incluirá uma parte fixa, uma parte extensível e as novas extensões, conforme mostra a Figura 12.

Assim sendo, a parte extensível da *UIL* é uma abstração interpretativa de uma ou mais linguagens computacionais subjacentes — por exemplo, a *EUPL*, as linguagens de programação usadas pelos designers de software, ou mesmo as linguagens *assembly* ou de máquina — se um usuário consegue entender completamente todos os signos da *UIL* devido:

1. Aos padrões de signos e combinações de signos que ele encontra enquanto interage com a aplicação;
2. Às explicações disponíveis sobre a *UIL* — da forma como estão disponibilizadas na *UEL* por meio de *help online* e tutoriais, *screen tips*, e documentação em geral da aplicação; e
3. À sua própria experiência com computadores — trazendo à tona o conhecimento associado usado nas metáforas da interface, conhecimento do domínio da aplicação e puro senso comum.

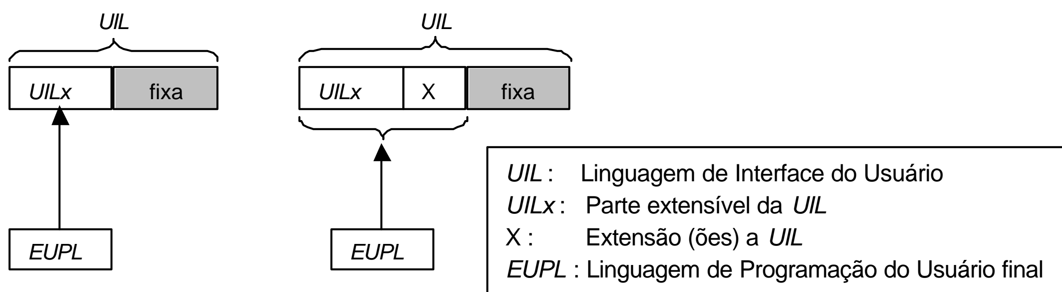

Figura 12: Mudanças na *UIL* como resultado de uma extensão.

Portanto, para entender a linguagem de interface o usuário não deverá precisar conhecer coisa alguma sobre as linguagens abstraídas. Todos os signos na linguagem de abstração — a *UIL* — devem satisfatoriamente fazer sentido para o usuário por meio das inferências que possam ser realizadas sobre o total de conhecimento que integram os itens 1, 2 e 3 acima, conforme mostra a Figura 13. Assim, formalmente, teremos:

<u>Definição</u>: Dado um conjunto de linguagens computacionais quaisquer L_i, L_j, ... L_n, L_i **é uma Abstração Interpretativa** de L_j, ..., L_n se:

- A semântica de L_i pode ser descrita pela união de todas as sentenças em L_j, ..., L_n; e
- Um usuário de L_i pode compreender "todos" os signos (léxicos e frasais) de L_i recorrendo a no máximo três fontes fundamentais e possivelmente sobrepostas:
    a. O padrão intrínseco de ocorrência de tais signos num discurso situado de L_i — isto é, sua experiência interativa;
    b. Algum conhecimento metalingüístico extrínseco de L_i — isto é, a documentação do software; e

c. Sua bagagem cultural — isto é, sua alfabetização computacional e seu conhecimento geral.

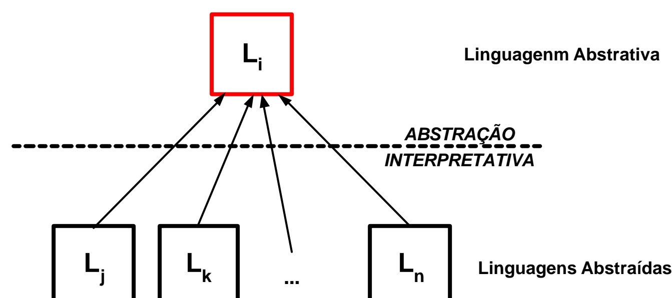

Figura 13: O princípio da Abstração Interpretativa.

Observe-se que o princípio de abstração interpretativa conta com as noções específicas de signo léxico e signo frasal. Um **signo léxico*** é uma palavra significativa de uma dada linguagem, enquanto um **signo frasal*** é um construto gramatical significativo de uma dada linguagem — isto é, uma estrutura organizadora de signos léxicos.

O princípio da Abstração Interpretativa é uma forma qualificada do princípio de abstração conhecido dos teóricos da informática. Este último enfatiza os mecanismos pelos quais detalhes são ocultados dos usuários, permitindo a eles focalizar os aspectos relevantes do domínio do discurso. Trata-se primariamente da interpretação do discurso situado produzido por sistemas computacionais (à medida que eles mostram aos usuários alguns resultados da aplicação) e pelos próprios usuários (à medida que eles dão entrada em comandos ou dados para o sistema). O aspecto gerativo do princípio de Abstração Interpretativa está moldado na maneira pela qual os usuários interpretam o que estão dizendo ao sistema, ou, em outras palavras, o que eles entendem por aquilo que estão dizendo.

O termo qualificador "interpretativa" é, desta forma, adicionado ao princípio já conhecido para fins de ênfase e clareza. Em um ambiente de *EUP*, todas as linguagens computacionais contribuem para as tarefas de design. Estas tarefas são dirigidas por intenções, e todas as intenções são determinadas não somente pela experiência dos usuários com o meio e o domínio de atividades — itens 1 e 3 acima —, mas também pela expressividade do código no qual esta experiência pode ser expressa — itens 1 e 2 acima. Assim, de fato, não é somente uma questão de abstrair as camadas semânticas subjacentes aos construtos sintáticos, mas também, e talvez ainda mais importante, de abstrair os objetivos e efeitos que podem ser alcançados tão logo os signos sejam

combinados. Logo, este princípio deve valer para as interfaces que tenham boa usabilidade em geral, e não somente para os ambientes de *EUP*.

2.2. O princípio do Contínuo Semiótico

No modelo de uso de software extensível descrito na Seção 1 deste capítulo assinalamos a necessidade de controlar o tipo do texto (extensão) que um usuário final pode gerar por meio da *EUPL*. Este controle é necessário para poder identificar os textos da *EUPL* que são válidos na *UIL*, isto é, os textos que são pragmaticamente adequados, e para garantir o Ciclo Mínimo de Interação do software. Ele também indica a existência de um forte relacionamento entre a *EUPL* e a *UIL* no software extensível. Tal relacionamento requer que os elementos da *UILx* possam ser expressos na *EUPL* (para que o usuário possa manipulá-los) e que os elementos expressos na *EUPL* tenham um reflexo na *UIL* (para que se constituam em uma extensão). O **princípio do Contínuo Semiótico**, proposto pela Engenharia Semiótica [DE SOUZA '01], procura avaliar o quanto dois códigos artificiais apresentam obstáculos à sua tradução mútua. Desta forma, ele avalia os obstáculos para traduzir extensões criadas na *EUPL* em construções funcionais e usáveis da *UIL*. Deste modo, este princípio serve como balizador da qualidade dos códigos empregados no software extensível.

Assim, duas linguagens computacionais são semioticamente contínuas [DE SOUZA '01] se seu acoplamento pragmático for sempre preservado ao se traduzir um texto de uma linguagem para outra em uma direção. No caso de aplicações extensíveis, a parte extensível da *UIL* (*UILx*) é Semioticamente Contínua com a *EUPL* se:

1. Os usuários em geral conseguem extrair sentido dos signos da *UILx* e interagir com a *UIL* sem qualquer conhecimento da *EUPL* ou de sua existência;
2. Ao estender aplicações, os usuários são capazes de compreender os signos da *EUPL* e gerar textos na *EUPL* sem qualquer conhecimento de outras linguagens de mais baixo nível, tais como linguagens de programação, as *APIs* de sistema computacional, etc.;
3. Existe um constituinte sintático para um "texto pragmaticamente válido" na *EUPL*; e
4. Qualquer usuário que conheça a *UILx* e a *EUPL* sempre pode traduzir uma instância arbitrária de um texto da *EUPL* para uma combinação válida, realizada ou potencial, de signos da *UILx*.

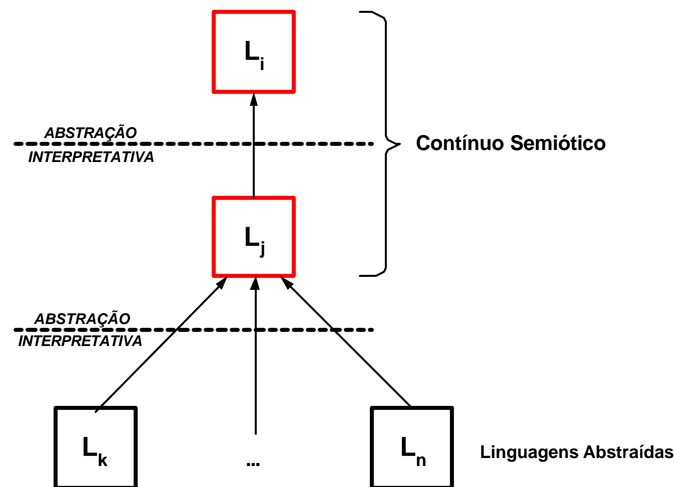

Figura 14: O princípio do Contínuo Semiótico.

Podemos ver que o princípio do Contínuo Semiótico procura atingir a exclusão das linguagens candidatas a *EUPL* que não distinguem combinações de signos pragmaticamente inadequadas das que são pragmaticamente adequadas. Este acoplamento pragmático entre a *EUPL* e a *UIL* é precisamente a característica que garante a correspondência entre textos bem-formados na *EUPL* e textos na *UIL*. Assim, formalmente temos:

Definição: Dadas duas linguagens computacionais L_i e L_j, elas são ditas **Semioticamente Contínuas** se (Figura 14):

- L_i é uma Abstração Interpretativa de L_j — isto é, L_j provê descrições semânticas de L_i;
- L_j é por si mesma uma Abstração Interpretativa de alguma(s) outra(s) linguagem(s) — isto é, a semântica de L_j é definida em alguma outra linguagem;
- L_j pode gerar instâncias de "texto", uma organização sintática especificamente estruturada de sentenças cujo significado incorpora elementos intencionais — isto é, existe um marcador sintático para a adequação pragmática dos textos de L_j em termos de L_i; e
- Qualquer usuário que conheça L_i e L_j **sempre** pode traduzir uma instância arbitrária de **texto** em L_j em uma combinação válida de signos, realizados ou potenciais, de L_i — isto é, não existe nenhum texto pragmaticamente adequado em L_j que não possa ser formulado em L_i.

Observemos a distinção feita entre signos realizados e potenciais. Um **signo realizado**[*] em uma linguagem é uma palavra ou frase existente nesta linguagem. Assim, quando usamos L_j para descrever signos realizados na L_i, existirá uma relação do mundo-a-palavra no direcionamento de adequação, conforme proposto por Searle [SEARLE ’79]. Por outro lado, um **signo potencial**[*] em uma linguagem é uma palavra ou frase não-existente nesta linguagem que, no entanto, pode ser gerada por extensões léxicas e/ou gramaticais que obedeçam aos seus padrões derivacionais — isto é, as suas meta-regras morfológicas e gramaticais. Logo, quando usamos L_j para gerar signos potenciais em L_i, estamos seguindo um direcionamento de adequação da palavra-ao-mundo.

Diferentemente do princípio de Abstração Interpretativa, que focaliza as linguagens de mais alto nível, este segundo princípio focaliza as linguagens de mais baixo nível. O ponto mais importante na definição acima é que, se alguém conhece ambos os códigos, ele ou ela sempre pode traduzir qualquer texto arbitrário da linguagem de mais baixo nível para um signo realizado ou potencial da linguagem de mais alto nível. Isto somente pode ser obtido devido a um componente pragmático que é projetado sobre estruturas sintáticas — o <texto> constituinte. Ele pode separar signos que expressam intenções dos que não o fazem e, desta forma, resolver textos em linguagens de mais baixo nível que devem ter uma tradução na linguagem de alto nível dos que não devem.

É importante notar que, para que o código da *EUPL* obedeça completamente o princípio do Contínuo Semiótico, é necessário que ela contenha estruturas sintáticas que garantam o acoplamento pragmático entre ela e a *UIL* de forma a refletir o Ciclo Mínimo de Interação anteriormente discutido. O padrão sintático geral da estrutura de um texto em uma *EUPL* Semioticamente Contínua é apresentado na Figura 15. As regras gramaticais e as estruturas sintáticas que devem aparecer em uma *EUPL* real serão discutidas no Capítulo 4 e 5.

Os elementos importantes descritos na Figura 16 são os três maiores componentes da estrutura do “texto da *EUPL*” que correspondem ao Ciclo Mínimo de Interação: a <mensagem anterior do sistema para o usuário>, a <ativação da ação> e a <mensagem subseqüente do sistema para o usuário>. Os parâmetros contextuais compartilhados por estes componentes (representados pelos índices que aparecem junto aos componentes) sustentam a consistência e **coerência**[*] do discurso neste trecho mínimo de interação humano-computador, que pode ser caracterizada pelas seguintes regras sintáticas:

```
<texto da EUPL>i,j ::=   <mensagem anterior do sistema para o usuárioi,
                         <ativação da ação>i,j,
                         <mensagem subseqüente do sistema para o usuário>i,j
<ativação da ação> i,j::= <mensagem de entrada do usuário>i,
                          <função ativada>i,j
```

Figura 15: *EBNF* para a gramática de alto nível de um texto na *EUPL*.

Assim sendo, para facilitar a criação de extensões pelos usuários finais é necessário que os códigos empregados na linguagem de extensão e na linguagem de interação do software extensível tenham um relacionamento muito bem definido. Por conseguinte, outro requisito a todo software extensível é que:

> *Os códigos nele presentes respeitem os princípios de Contínuo Semiótico e da Abstração Interpretativa de forma a facilitar sua aquisição e utilização pelos usuários finais.*

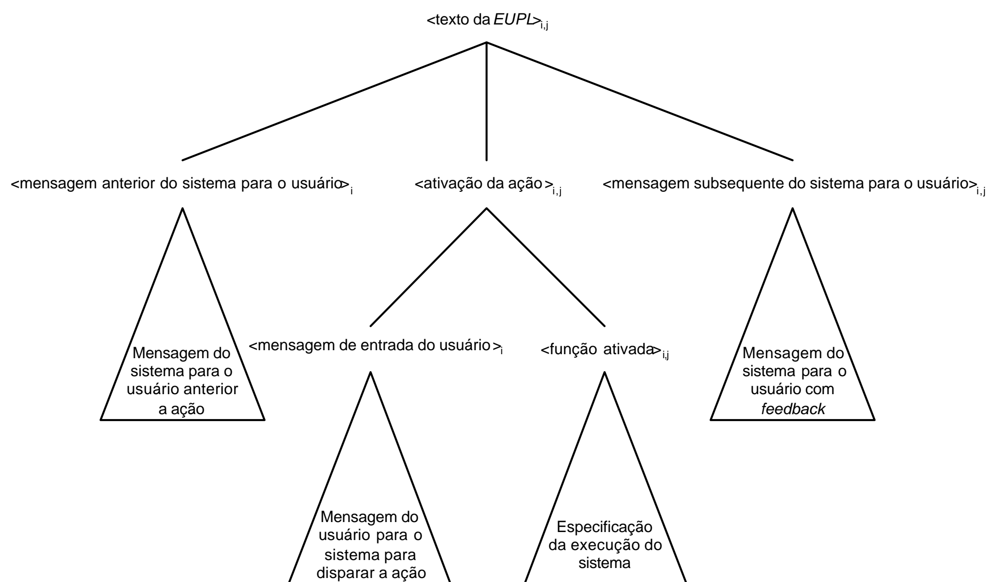

Figura 16: Um exemplo da estrutura do texto requerida pelo princípio do Continuo Semiótico.

A relação de continuidade semiótica entre as linguagens presentes em um software extensível pode ser resumida no diagrama apresentado na Figura 17. Nele podemos ver que a *UEL* deverá ser semioticamente contínua tanto com a *UIL* quanto com a *EUPL*. Esta dupla relação deve existir para que se possa satisfazer os critério de usabilidade definidos por Adler e Winograd.

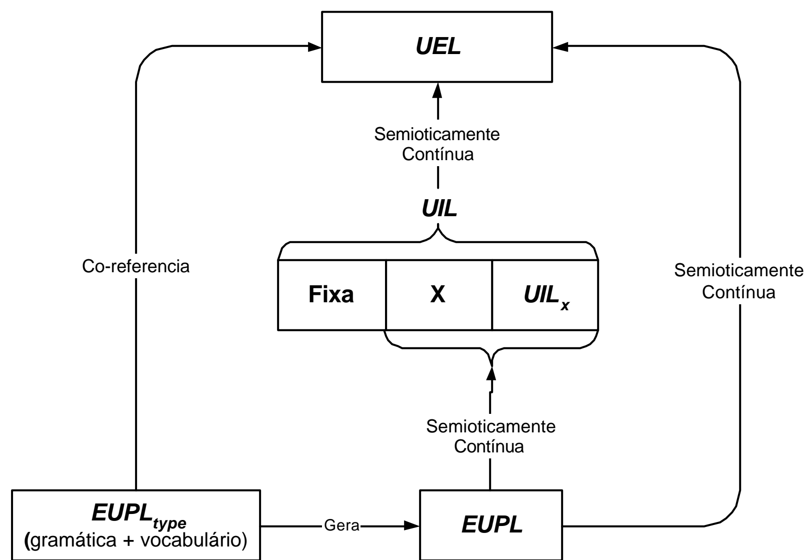

Figura 17: Descrição da relação de Continuidade Semiótica entre a *UEL*, *UIL*, e a *EUPL* de um software extensível.

Com a definição destes dois princípios, fechamos nossa análise sobre a natureza e o perfil dos códigos empregados no software extensível. Até o momento, levantamos os requisitos estruturais necessários à obtenção de um software extensível que apresente uma boa comunicabilidade de suas funções de forma a possibilitar a um usuário final criar suas próprias extensões. Resta-nos ainda definir uma linguagem de extensão que seja de fácil aprendizado e uso pelos usuários finais e um processo para a tarefa de *EUP* que oriente estes usuários nas etapas a serem seguidas.

Capítulo 4

UMA ANÁLISE DA LINGUAGEM DE PLANOS DAS PESSOAS

Neste capítulo, apresentamos uma análise lingüística da linguagem empregada por pessoas comuns (entenda-se não programadores) para expressar planos no dia-a-dia. O objetivo desta análise é subsidiar a definição de uma linguagem de extensão que seja de fácil aprendizado e uso pelos usuários finais. Nesta análise, avaliamos uma dezena de receitas e instruções de faça-você-mesmo encontradas na *Internet* e alguns outros tipos de manuais de instrução, procurando identificar pontos de semelhança e discrepância entre o uso da linguagem natural (LN) feito pelas pessoas para expressar seus planos e o uso das linguagens de programação feito por programadores na construção de programas. A escolha pela análise da linguagem de planos das pessoas é proveniente de uma observação feita por Linden, em um artigo sobre design automático de software [LINDEN '91], na qual ele nos alerta para o fato de que "programação e planejamento são atividades análogas". Ele diz:

> *"Programming and planning are analogous activities, ... At a high level of abstraction, planning and programming are the same. In both, we begin from a set of goals, initial conditions, and primitive operations and we want to produce a composition of operations that will accomplish the given goals."*

Assim, apresentaremos primeira seção uma série de considerações gerais que identificamos como relevantes para nosso trabalho e na segunda seção descreveremos a sintaxe do subconjunto da LN que é empregada pelas pessoas na expressão de planos no dia-a-dia.

1. Considerações gerais sobre a linguagem de planos das pessoas

A primeira observação geral que temos a fazer é a aplicação — pelas pessoas — de "conhecimento de senso comum" na resolução das ambigüidades que surgem na interpretação dos planos no dia-a-dia. Tais ambigüidades nascem do uso da LN na representação de seqüências de ações. Esta fonte de conhecimento, em sua totalidade, é vasta e difícil de capturar, no entanto, para domínios bem definidos, podemos criar um modelo limitado para ela.

Notamos também que quando existe uma necessidade de "instruções seguras" — sem ambigüidade — como ocorre em procedimento de emergência para aviões, são impostas restrições sobre a "forma da linguagem" que pode ser empregada para descrever tais instruções. Apesar de frouxas, estas restrições são similares às impostas aos programadores profissionais pelos compiladores e interpretadores de LPs. Elas limitam as interpretações permitidas para uma dada sentença, visto as formas escolhidas serem minimamente polissêmicas. Abordagem semelhante tem sido pesquisada para a especificação de requisitos na engenharia de software [FUCHS '96].

Uma segunda observação geral é que existem dois tipos de planos com os quais as pessoas estão acostumadas. Instruções operacionais e manuais de usuários são planos operativos — isto é, eles somente dão instruções sobre como usar uma funcionalidade específica de um sistema. Eles não modificam a natureza dos objetos presentes no sistema. Por outro lado, receitas e instruções de faça-você-mesmo são planos gerativos — isto é, eles dão instruções sobre como modificar a natureza dos objetos de um sistema. Eles normalmente criam, por meio de transformações, novos objetos no sistema ou modificam a constituição dos antigos.

A implicação direta destes dois tipos de planos é que devem existir dois tipos de linguagem para expressá-los. Planos operativos somente precisam de algum tipo "linguagem de manipulação".

Planos gerativos, por outro lado, precisam de alguma "metalinguagem" para possibilitar a criação de novas categorias de objetos nos modelos do domínio, ou para criar novas ações para os atuais objetos do domínio. Na verdade, os planos gerativos englobam os planos operativos, uma vez que é necessário manipular os objetos existentes para obter novos objetos.

Nossa terceira observação geral, e muito provavelmente a mais importante, é que o que torna as linguagens de planos das pessoas comuns única é a forma como elas manipulam os objetos do domínio. Suas linguagens usam mecanismos diferentes dos usados nas LPs atuais para fazer referenciar aos objetos. O seu mecanismo de referência apresenta uma forma natural de operar com a estrutura dos objetos por meio do emprego de conhecimento do senso comum, quantificadores, qualificadores e figuras de linguagem como anáforas, elipses, metáforas e metonímias. Tais mecanismos ajudam as pessoas quando elas se referenciam às características de objetos complexos, uma vez que eles ocultam muitos dos detalhes sobre a implementação de sua estrutura.

O tratamento de figuras de linguagem como, por exemplo, as anáforas e as elipses, não tem sido totalmente esquecido pela comunidade de LP. Um tipo de mecanismo de referência anafórica já se encontra presente na linguagem de roteiros[30] *Hypercard™* [HYPERCARD '93]. *Hypercard* enfatizou a legibilidade dos roteiros como uma de suas principais características. Ela fez uso de variáveis especiais tais como, ***it***, ***me***, ***next***, ***previous***, e ***this*** para permitir ao usuário referir-se a objetos de forma implícita. ***It*** é o destino para o resultado de um subconjunto dos comandos da linguagem podendo ser utilizado nos mesmos lugares que referências a objetos. ***Me*** é usado dentro de um roteiro para referenciar o objeto que contém o *handler* atualmente em execução. ***Next***, ***previous*** e ***this*** são empregados para referir-se, respectivamente, ao próximo, anterior ou atual cartão, background ou pilha de objetos.

O emprego destas variáveis especiais ampliou a legibilidade dos roteiros. Contudo, este mecanismo trata anáforas somente pela lembrança do último resultado de alguma ação especial. O que é mais importante, a utilização errônea de ***me***, do ponto de vista de uma análise pragmática do discurso [LEVINSON '83], força os usuários finais a trocar de papéis durante o discurso e a comportar-se tanto como um escritor de roteiro quanto como um tópico* do roteiro. Isto acontece porque ***me*** não se refere à primeira pessoa do discurso, mas ao objeto sobre o qual a pessoa está escrevendo.

---

[30] *Scripts*.

A quarta e última observação é que receitas e instruções de faça-você-mesmo capitalizam sobre a nossa capacidade para a categorização [LAKOFF '90] de modo a tornar o processo de transformação dos planos gerativos particularmente natural. Assim, o processo de transformação de um objeto é visto como um movimento suave e natural de uma categoria de objetos para outra, mesmo quando mais de um objeto é transformado. Este movimento entre as categorias é baseado em mudanças nas características do objeto original, as quais podem ser estruturais ou comportamentais.

2. Uma análise lingüística da linguagem de planos das pessoas

Tendo estas observações gerais em mente, vamos agora analisar mais cuidadosamente cada elemento da linguagem de planos das pessoas comuns. Uma vez que os planos gerativos englobam os operativos, nós analisaremos somente os primeiros. Sempre que for útil à nossa analise, também levaremos em consideração exemplos de planos operativos. Deste modo, concentraremos nossas atenções em receitas e instruções de faça-você-mesmo. Primeiramente, discutiremos os elementos conceituais presentes no subconjunto da LN que estamos considerando e, então, mostraremos uma análise lingüística das estruturas presente neste subconjunto.

2.1. A composição da linguagem

Olhando para texto de receitas culinárias encontramos que o que, normalmente, vem em primeiro lugar é a lista dos ingredientes. Ingredientes são objetos — matéria a ser transformada e recursos necessários para transformá-la. É comum existirem condições definidas sobre o estado inicial dos objetos e/ou sobre toda a receita (por exemplo, o uso de estruturas qualificadoras tais como adjetivos —"*1/8 teaspoon of ground red pepper (cayenne)*"—, sentenças relativas e congêneres).

Após os ingredientes, vêm os passos operativos e/ou gerativos. Tipicamente, uma representação textual de um passo é uma sentença composta. Cada passo pode ser composto de várias ações que são representadas por sentenças simples. Assim, se nós lermos uma representação textual focalizando estritamente na sua estrutura sintática, não haverá uma forma clara de distinguir entre uma seqüência de passos e um único passo. Quando estivermos fazendo o *parsing* de um conjunto de sentenças nesta linguagem, poderá ser possível gerar um passo com muitas ações ou muitos passos com um número variável de ações. Deste modo, é difícil diferenciar o fim de uma ação do fim de um passo usando critérios estritamente sintáticos. A solução será considerar

marcadores retóricos ou discursivos e montar as estruturas sintáticas dentro da estrutura do discurso. Para sinalizar esta solução em representações textuais, freqüentemente, fazemos uso de redundância tipográfica — tais como endentação, listas, formatação de parágrafos etc. — como pista visual. Por exemplo, um parágrafo sempre descreve uma ação gerativa sobre um único objeto[31].

Outro ponto importante é que não é comum termos um bloco de instruções a ser especificado, como acontece nas LPs. Na maioria das vezes, o bloco está implícito. Por outro lado, é comum ter rótulos marcando os passos, e ir de um passo para outro usando uma instrução semelhante a um "*go to <label-name>*". Este mecanismo pode ser empregado para corrigir erros detectados na execução de um passo, ou na criação de repetições dentro do plano.

Em alguns casos, podem existir descrições de sub-planos — partes de um plano que podem ser executadas: antes, depois ou paralelamente a outras partes. Normalmente, estes sub-planos são chamados diretamente sem "parâmetros". Quando existirem parâmetros, eles podem estar implícitos, referenciando os objetos que pertencem ao contexto atual do passo. Estes parâmetros implícitos são, de fato, o resultado do rastreamento do **foco do discurso**[*], o conjunto de objetos de que se está falando e, portanto, não precisam ser referenciados explicitamente. Esta noção também é importante para ajudar a resolver referências anafóricas e elípticas. Deste modo, parâmetros podem ser tanto uma referência a um objeto ou o resultado da execução de outro sub-plano. Algumas vezes, o último caso cria uma ambigüidade referencial potencial, uma vez que se torna difícil distinguir entre o fim de uma lista de parâmetros e o fim de uma lista de ações.

Um outro ponto observável é a não existência de uma distinção clara entre uma ação primitiva e um sub-plano. Isto acontece devido ao alto nível da especificação das ações primitivas, uma vez que os verbos usados em LN para expressá-las são interpretados pelas pessoas empregando-se conhecimento do senso comum. Além disso, é importante ressaltar que "uma ação é sempre aplicada a um objeto" — isto é, sempre existe um objeto que se submeterá aos efeitos de uma ação. Uma ação somente aparecerá se ela produzir uma transformação no estado do sistema — isto é, "não existe ação nula". Ações usualmente têm pré-condições sobre o estado dos objetos sobre os quais elas irão atuar. Além disso, é comum usar conjunções e disjunções na descrição destas pré-condições e na descrição das ações dos passos. É digno de atenção o fato de que a

---

31 Um parágrafo compreende um passo, e um passo cria ou modifica somente um objeto, por meio de outros passos e pela manipulação de objetos.

ordem sintática de aparecimento dos objetos e das ações não é fixa — isto é, normalmente os objetos seguem as ações, mas algumas vezes o inverso ocorre. Esta troca na ordem é, normalmente, causada por **restrições focais** que dão suporte a **coesão do discurso** na LN. Este fato, juntamente com a forma como as características dos objetos são especificadas, são pontos que fazem o mecanismo de referência desta linguagem mais flexível que os disponíveis nas LPs atuais.

É extraordinário que, com respeito às transformações de objetos, à primeira vista não parece haver noção direta de variável. Os objetos são simplesmente declarados no começo do plano, e têm sua funcionalidade ou propriedades referenciadas ou empregadas por todo o plano. Numa visão mais próxima, no entanto, vemos que o que realmente ocorre é que algumas palavras referenciam *types* — uma classe inteira de objetos — e, portanto, simbolizam variáveis. Isto é possível devido ao emprego de mecanismos anafóricos específicos. Na maioria dos casos, tais mecanismos permitem trocar um artigo indefinido — o qual denota uma variável universalmente quantificada (assim como em "*an egg*") — por um artigo definido — o qual pode denotar um objeto único específico (assim como em "*the egg*"). Os substantivos que aparecem na parte de declaração dos planos denotam *types*, podendo ser precedidos por quantificadores. Quando os mesmos substantivos aparecem no corpo do plano, eles denotarão, geralmente, um objeto unicamente definido ou um conjunto — um *token* —, e deverão ser precedidos por artigos definidos. Esta é uma diferença crítica entre os planos das pessoas comuns e a programação nas LPs atuais. Palavras no corpo de um plano normalmente referem-se a um *token*, mas em alguns casos, elas podem referir-se a um *type*. Por exemplo, no texto "*In the document, if the paragraph style is equal to 'Heading 1' then go to the beginning of the paragraph and break the page*", a palavra "*paragraph*" representa todos os parágrafos do documento em vez de um só. A diferenciação entre *token/type* nas linguagens de planos das pessoas comuns é sutil e sensível ao discurso.

Adicionalmente, com referência à transformação de objetos, quando uma transformação degrada as características do objeto, a estratégia mais comum é dar a ele um novo nome, como uma forma de especificar que ele agora pertence a uma nova categoria. Este processo de determinar quando o objeto precisa receber outro nome faz parte do mecanismo categorização. Além disso, quando um conjunto de objetos é transformado no seu total, o objeto resultante também recebe um novo nome e, normalmente, se cria uma nova categoria.

O fator mais importante no nível da linguagem de manipulação é sua expressividade. Esta linguagem é, na sua maior parte, constituída pelo mecanismo de referência, anteriormente

mencionado, e pelas ações operativas. A parte operativa tem "condicionais" como em "*If too stiff to spread, add a small amount¼*"; e iteradores como em "*Beat until of spreading consistency.*"[32] Não existe interações do tipo *while*, utilizadas com o mesmo sentido que nas LPs atuais, nas linguagens de plano empregadas pelas pessoas. Existem duas razões para isto:

- Na expressão de planos do dia-a-dia, as pessoas sempre pensam em ações realizáveis[33] — isto é, ações que tenham algum efeito no mundo. Uma ação que não resulta em uma transformação no mundo não necessita ser expressa. Por esta razão, não existe a noção de repetição "nula" nas linguagens de plano destas pessoas. Isto descarta as iterações *while* no sentido que elas são usadas nas PLs atuais, e
- Nas linguagens de planos das pessoas comuns, *while* é, normalmente, usado para expressar paralelismo como em "*In a…cook the onion with…stirring, until the onion is golden. While the onion is cooking…*". Nesta sentença, vemos que as cebolas já devem estar cozinhando quando o *while* é alcançado e, portanto, podemos usar o tempo livre "*while*" (enquanto) elas estão cozinhando para começar uma ação paralela. Deste modo, o condicional existente no interior do comando *while* sempre será satisfeito diretamente e, novamente, não ocorrerá uma iteração nula.

A linguagem de manipulação também permite a aplicação de ações[34] com e sem parâmetros, como discutido anteriormente. No entanto, não existe uma noção direta de atribuição, uma vez que não existe uma noção direta de variável. Nas LPs atuais, uma atribuição muda o valor do objeto preservando somente seu nome e classificação, o que não ocorre nas LNs. O que encontramos nos textos em LN é a mutação implícita do objeto (isto é, um objeto tendo algumas de suas características mudadas devido a uma transformação que o próprio objeto sofre), como em "*Take the onions, garlic and olive oil, and add them to the mix in the bowl. The sauce is now ready to serve*". Um mecanismo de categorização deverá, portanto, ser capaz de cuidar da fusão dos objetos conhecidos (tais como *onions*, *garlic* e *olive oil*) em novos objetos (tais como *sauce*).

Junto com estes comandos básicos, algumas ações podem ter sua duração especificada. Existem diferentes formas de fazermos isto. Podemos usar uma expressão "*while…*" ou "*or,*

---

[32] *Until* é normalmente empregado para repetir uma ação ou um pequeno conjunto de ações.

[33] Seguindo as máximas da comunicação de Grice [GRICE ´75].

[34] ***Actions***.

until…" quando o tempo não é precisamente especificado, ou um "*for…*(unidades de tempo)" e "*about…*(unidades de tempo)" para especificar uma duração específica. Sempre que uma ação é muito longa e não pode ser interrompida por e/ou coordenada com outra, ações paralelas podem ser iniciadas. Apesar da necessidade de expressarmos a duração ser central para aplicações de tempo-real, ela não é tão crucial assim para o tipo de aplicações que estamos considerando neste trabalho. Logo, não entraremos em mais detalhes sobre as forma de expressar o tempo neste trabalho.

2.2. A estrutura da linguagem

Vamos agora apresentar uma análise deste subconjunto da LN procurando identificar as principais estruturas que o compõem. A seguinte notação será empregada para expressar os padrões de sentenças na linguagem: **S** - Sentença, **V** - Verbo, **O** - Objeto, e **A** - Adjunto. Usaremos também "/" para separar as partes de uma sentença e ilustraremos os casos dos adjuntos com um subscrito fechado por "**[ ]**".

Podemos identificar dois grupos principais de sentenças imperativas no *corpus* analisado. Um grupo de "formas básicas" que são dadas pelas seguintes estruturas:

- **S → V O**

  Add / shallots and oregano.

- **S → V O A**

  Fill / cup / with pink cellophane grass $_{[instrument]}$.

  Trace / the template / in a piece of paper $_{[place]}$.

- **S → V O A A**

  Boil / sauce / over high heat $_{[manner]}$ / until heated through, about 5 minutes $_{[duration]}$.

  Secure / the buttons / to the flowers $_{[place]}$/ by threading a 24-gauge wire through them $_{[manner]}$.

E um grupo de formas complexas, que apresentam transformações **usadas para fins retóricos** de manutenção do **foco do discurso**. Algumas destas transformações são obrigatórias, quando usadas para evitar a quebra da coesão do texto; enquanto outras são opcionais, quando usadas para enfatizar um certo elemento na sentença. Estas formas são dadas pelas seguintes sentenças:

- **S → A , V O**

  On a copy machine $_{[means]}$, / make / eight copies.

Using tongs [instrument], / turn / chicken over.

- **S → A , V O A**

Using the bristle brush [instrument], / base coat / the door / Dusty Beige [manner].
For a protective finish [goal], / apply / Matte Varnish / with the sponge applicator [instrument].

Como pode ser visto pelos exemplos, as formas complexas são construídas pela inversão da posição de uma classe limitada — os adjuntos — em relação aos verbos. Esta inversão não muda a semântica do verbo, mas amplia os seus aspectos comunicativos (os aspectos retóricos) da sentença.

Nossa análise mostrou também que existe um emprego comum de elipses para ocultar os objetos das sentenças dentro de um parágrafo. Isto evita a necessidade de escrever e, portanto, de interpretar sentenças muito longas, o que é particularmente trabalhoso para as pessoas. Este efeito é confinado às sentenças das formas básicas e produz como resultado as seguintes sentenças:

- **S → V ( ) A**

… Sand / (it) / lightly [manner].
… Sauté / (them) / until skin browns, about 5 minutes [duration].

- **S → V ( ) A A**

… Bolt / (it) / on the palette [place]/ so the sponge is wet with paint [manner].
… Place / (them) / in water [place]/ to expand [goal].

Nestas sentenças o pronome oculto está se referindo a um objeto já mencionado no contexto formado pelas sentenças anteriores. Deste modo, ele pode ser omitido sem causar qualquer problema de interpretação. Este tipo de mecanismo está ausente nas PLs atuais e força os programadores a escrever um código muito detalhado, o qual usualmente tem uma comunicabilidade muito baixa e, portanto, é muito difícil de ser lido e compreendido.

As estruturas mostradas acima não são as únicas no subconjunto da LN analisado, contudo, são as mais comuns no contexto dos planos analisados. Elas podem ser resumidas pela seguinte regra gramatical em *EBNF*:

- **S → [A ‘,’] V [O] [A [A]]**

Estas estruturas estão relacionadas principalmente à chamada de um procedimento nas LPs atuais. Sua semântica é comparável à da abordagem funcional para as LPs, onde os parâmetros são

especificados pelos adjuntos (**A**). As diferenças mais relevantes entre estas estruturas e as LPs atuais estão na possibilidade de omitir o objeto — usando elipses —, de inverter a posição de um parâmetro — um adjunto — e de usar anáforas — um tipo bem restrito de ponteiros — para fazer referências a objetos. Estes mecanismos são empregados para ampliar a qualidade da comunicação nestas linguagens e não apresentam equivalente na LPs atuais.

A estrutura geral dos adjuntos varia significativamente. Os adjuntos mais comuns neste subconjunto de LN são os de posição, processo — instrumento, meio e modo —, objetivo e duração. Todos eles representam parâmetros para a ação expressa pelo verbo que os rege. Em geral, a estrutura de um adjunto é composta de uma preposição (**Prep**) seguida de um objeto (**O**), mas ela também pode ser um sintagma verbal ou um sintagma nominal. Levaremos em consideração somente as forma mais comuns [BIBER'99], o que nos deixa com a seguinte regra gramatical:

- **A → Prep O**

Cada tipo de adjunto tem seu próprio conjunto de preposições, no entanto algumas preposições podem agregar-se a mais de um adjunto, tornando difícil empregar somente as preposições para distingui-los.

A estrutura geral dos objetos é particularmente complexa e emprega um intrincado conjunto de regras de restrições sobre os elementos presentes na sua forma final [QUIRK '72]. Tal estrutura nos fornece um conjunto muito rico de possibilidades para referenciar objetos dentro do domínio. Toda esta flexibilidade não é, realmente, requerida na área de *EUP*, contudo, quanto mais disponibilizarmos, mais facilmente os usuários finais se expressarão. Podemos usar as seguintes regras gramaticais, simplificadas e sem considerar as restrições, em *EBNF* para os objetos:

- **O → PrM N ['of' PrM N]**
- **PrM → [PrD] D [[Ord] [Card] [N]]**

Onde **PrM** - pré-modificador, **N** - substantivo, **PrD** - pré-determinante, **D** - determinante, **Ord** - ordinal, e **Card** - cardinal.

Os pré-determinantes — ***all*** —, os determinantes — ***each*** e ***every*** — e os substantivos plurais descrevem a maioria das iterações na LN. Estas iterações estão implícitas dentro da

referência aos objetos e, normalmente, ocorrem sobre a hierarquia de relações todo-parte que existe entre os elementos do domínio. Os ordinais são normalmente empregados para o caminhamento dentro das estruturas no domínio. Este tipo de caminhamento é natural para as pessoas e não ocorre nas LPs atuais. Um determinante e um substantivo podem ser substituídos por pronomes de vários tipos. Neste caso, eles são normalmente usados como anáforas, permitindo um emprego convenientemente controlado e seguro de ponteiros na LN.

Com estas observações, concluímos nossa análise do subconjunto da LN que compreende a linguagem de planos das pessoas comuns. Esta análise nos possibilita definir um conjunto de características desejáveis a uma *EUPL* para que ela apresente boa comunicabilidade.

Assim, podemos concluir que o emprego de uma linguagem de extensão baseada em planejamento é um bom caminho para aproximar o usuário final de um ambiente de extensão sem, no entanto, obrigá-lo a aprender um grande conjunto de conceitos específicos da área de computação. Por outro lado, será necessário delimitar muito claramente o escopo desta linguagem. Primeiro, para que o usuário não presuma que a LN inteira esteja disponível para ser usada, e, segundo, para manter seu tratamento computacional dentro de uma faixa de eficiência razoável, de forma a não inviabilizar sua implementação.

Capítulo 5

UMA LINGUAGEM-TIPO PARA *EUP*

Neste capítulo, apresentamos uma linguagem-tipo para extensão por usuários finais baseada no Modelo Semiótico para a tarefa de *EUP* descrito no Capítulo 3 e na análise da linguagem natural empregada pelas pessoas no seu dia-a-dia para expressar planos apresentada no Capítulo 4. Na primeira seção, descreveremos o tipo de aplicação que pode ser estendida por este tipo de linguagens e, com isso, discutiremos quais mudanças necessitam ser incorporadas às arquiteturas de software atuais de modo a produzir um software extensível de melhor qualidade. Na segunda seção, apresentaremos uma descrição da linguagem-tipo para *EUP* proposta de modo a caracterizar as *EUPLs* que seguem este modelo. O diferencial desta linguagem-tipo encontra-se na inclusão de mecanismos lingüísticos presentes no subconjunto da LN identificados na análise do Capítulo 4 e que não se encontram presentes nas linguagens de programação (LP) atuais. Discutiremos, ainda, como realizar o projeto de uma linguagem semelhante à proposta, abordando os mecanismos necessários à interpretação de uma linguagem deste tipo. Trataremos também da forma da linguagem de representação de conhecimento (*KRL*) necessária à criação e manutenção de um modelo do software que possibilite a manutenção do Contínuo Semiótico entre a *EUPL*, a *UILx* (a parte extensível da *UIL*) e a linguagem de explicação para o usuário final (*UEL*) de uma aplicação extensível.

1. Aplicações extensíveis de software

No Capítulo 3, descrevemos um modelo teórico para a tarefa de *EUP* baseado na teoria Semiótica e no Modelo de Comunicação Verbal de Jakobson. A visão do software como uma meta-mensagem e do seu uso como um processo de comunicação nos levou a propor um conjunto de características que contribuem para a usabilidade deste tipo de software. Estas características, aliadas a testes realizados com a versão preliminar de uma instância da linguagem-tipo para *EUP* aqui proposta (apresentados no Anexo I), serviram de base para a definição de um processo para a tarefa de *EUP*, que será apresentado no Capítulo 6. É importante salientar que a *EUPL* de uma aplicação extensível deve estar intimamente ligada ao processo para a criação de extensões que seu modelo teórico preconiza. Deste modo, a linguagem-tipo para *EUP* aqui descrita foi desenvolvida concomitantemente ao processo para a tarefa de *EUP* preconizado pelo Modelo Semiótico. Optamos por apresentar primeiro a linguagem-tipo por crermos que assim facilitamos a compreensão do Modelo Semiótico como um todo.

Muito embora as arquiteturas de software atuais façam bom uso dos recursos de hardware e do sistema computacional, de maneira geral, elas infelizmente ainda não são capazes de fornecer suporte adequado às aplicações que pretendam possibilitar sua extensão por parte de usuários finais. Além do **mecanismo de extensão** que precisa necessariamente ser adicionado, elas carecem de bons **mecanismos de explicação** (help) e de **tratamento de erros**. Tais mecanismos são, como descrito no Capítulo 3, de fundamental importância para que um usuário possa adquirir um bom conhecimento sobre o funcionamento do software e, assim, venha sentir a necessidade de estender sua funcionalidade.

Deste modo, a disponibilização de uma ***UEL*** **sensível ao contexto de uso** e **evolutiva** — isto é, capaz de continuar operativa mesmo após a realização de extensões ao software — é requisito essencial às aplicações de software que implementam o Modelo Semiótico, conforme descrito no Capítulo 3. Tal linguagem será a base do mecanismo de explicação que possibilitará ao usuário esclarecer suas dúvidas sobre o modelo de usabilidade e extensibilidade do software. Moore [MOORE '95] nos mostra que para termos um mecanismo de explicação que possa responder de forma adequada às situações individuais de uso, é necessário que sejam empregadas várias formas de conhecimento na geração do texto explicativo. Ela descreve ser necessário manipular conhecimento dos seguintes tipos:

- **Ontológico**, a ser empregado na descrição dos elementos do domínio e na representação das restrições de seus relacionamentos;
- De **design do software**, a ser empregado na explicação das estratégias de resolução de problemas utilizadas no software (que incluem sua interação com usuário); e
- **Lingüístico**, a ser empregado na interpretação das perguntas do usuário e na geração do texto explicativo.

No entanto, é importante lembrar que o usuário de um software extensível interage principalmente com elementos da *UIL* e da *EUPL*, somente recorrendo à *UEL* quando não consegue interpretar algum destes elementos ou quando ocorre um erro na execução da tarefa corrente. Assim sendo, para que o mecanismo de explicação possa funcionar de forma adequada, é preciso garantir que estas linguagens se inter-referenciem nestes vários planos de representação, o que significa ter uma **representação lingüística comum** para os elementos do domínio e para as regras de design empregadas pelo designer do software, a qual será manipulada por estas três linguagens. Formas semelhantes de representação são empregadas em outros tipos de mecanismos utilizados na criação de extensões como, por exemplo, as *data descriptions* em PD [CYPHER '93A]; e também são indicadas como sendo necessárias aos agentes de software que atuam como auxiliares dos usuários na realização de suas tarefas [LIBERMAN '98]. Além disso, esta representação lingüística comum deverá dar suporte às formas de conhecimento descritas por Moore, de modo a gerar um texto adequado a cada situação de uso e de erro do software.

Trabalhos anteriores [DERTOUZOS '92] [SMITH '92] já sugerem que as arquiteturas de software extensível devem ser modulares, de modo que um usuário possa agregar novos componentes ao seu software configurando-o às suas reais necessidades, o que não é, normalmente, respeitado nas arquiteturas de software atuais. Em vista disso, uma nova arquitetura está sendo proposta pelo *SERG* para dar suporte ao software extensível [DE SOUZA '97]. Esta arquitetura contempla os requisitos levantados no Capítulo 3 e as observações de Moore. Na Figura 18, apresentamos uma versão desta arquitetura, revisada para incluir os mecanismos resultantes dos requisitos identificados neste trabalho. Ela consta basicamente dos seguintes elementos:

1. Um ***back-end* de execução** da aplicação — que executará a funcionalidade do software, sendo equivalente ao existente nas arquiteturas de software atuais;

2. Uma **interface multimodal e multicódigo** — que compõe a *UIL* do software (também presente nas arquiteturas de software atuais);
3. Uma **base de conhecimento do design da aplicação** (*ADKB*) — que contém os modelos do domínio e do software e os mecanismos de raciocínio necessários à sua utilização e manutenção.
4. Um **sistema de explicação** — composto por um agente de software responsável pela interpretação das perguntas do usuário e pela geração de textos na linguagem de explicação;
5. Um **sistema de tratamento de erros** — composto de um agente de software operando em conjunto com o sistema de explicação, de modo a tornar o tratamento de erros sensível ao contexto de ocorrência do erro; e
6. Um **sistema de extensão** — composto por um conjunto de agentes de software que serão responsáveis pela interpretação da linguagem de extensão — a *EUPL* — e pela manutenção do Contínuo Semiótico entre a *EUPL*, a *UILx* e a *UEL*.

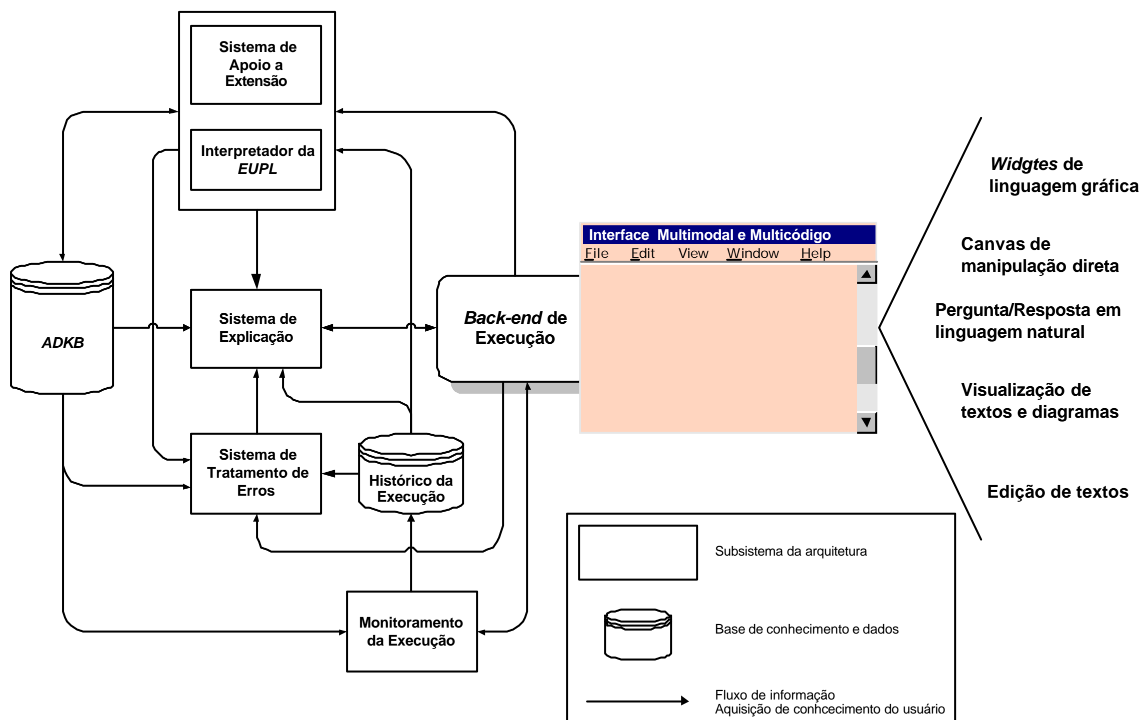

Figura 18: Arquitetura de software proposta no *SERG* para software extensível.

Os dois primeiros elementos, o *back-end* de execução e a interface multimodal, são elementos comuns nas aplicações atuais. O *back-end* dispensará mais comentários, visto que ele

poderá ser implementado utilizando as técnicas correntes da engenharia de software. A interface que compõe a *UIL* requer mais cuidados dentro da abordagem da Engenharia Semiótica, porém, ela não será alvo direto de estudos neste trabalho. Várias outras pesquisas foram desenvolvidas no *SERG* visando à identificação de diretrizes para a definição da *UIL* do software empregando a abordagem da Engenharia Semiótica [LEITE '98] [PRATES '98] [MARTINS '98] e os resultados destas pesquisas serão aproveitados neste trabalho.

Mantida a organização proposta na Figura 18, podemos dizer que são os elementos de 3 a 6 que diferenciam uma aplicação extensível que siga o Modelo Semiótico proposto no Capítulos 3 das que não o seguem. Estes elementos são essenciais para capacitar um usuário final à criação de extensões e também à manutenção do Contínuo Semiótico entre as linguagens presentes num software extensível.

A presença de uma *ADKB* em um software extensível é primordial para manter uma representação comum dos elementos do domínio, funcionando, portanto, como uma **interlíngua** entre as linguagens usadas pelos usuários finais num software extensível — a *UIL*, a *UEL* e a *EUPL*. Assim, ela será empregada na manutenção do relacionamento entre os diferentes códigos utilizados no software extensível e, portanto, será a responsável por garantir a Continuidade Semiótica entre eles. Como função secundária, ela servirá de fonte de conhecimento para a geração de explicações na *UEL* sobre o funcionamento da aplicação. Ela estará dividida em dois componentes[35]:

- Um **modelo do domínio** — que represente a natureza intrínseca entre os objetos do domínio (suas relações estáticas) e as interações, conforme vistas pelos usuários finais do software, entre estes objetos no tempo e para um fim particular (suas relações dinâmicas).
- Um **modelo do software** — que represente as interações entre alguns objetos do domínio e os objetos do sistema computacional (necessários à implementação da funcionalidade do software e de sua interface). Neste modelo, deverão estar representadas também as regras de design empregadas pelo designer do software para sua construção. Estas regras serão empregadas para guiar a construção dos elementos de interface necessários à criação de extensões.

---

[35] Payne [PAYNE '91] apresenta os mesmos componentes sob uma perspectiva diferente em sua discussão sobre uma abordagem ecológica para modelagem da interação humano-computador.

O primeiro componente modelará a organização do domínio da aplicação (relacionando-se ao conhecimento ontológico de Moore) e o segundo modelará a solução desenvolvida pelo designer para os problemas do usuário (relacionando-se ao conhecimento de design do software descrito por Moore). Assim, estes componentes perfazem somente os dois primeiros tipos de conhecimento especificados por Moore. O conhecimento lingüístico, a terceira forma, estará embutido nos agentes de software empregados para a interpretação das perguntas dos usuários e para a geração de texto realizada no sistema de explicação que conterá a linguagem de explicação.

Para que o **sistema de explicação** faça bom uso de uma *UEL* que seja realmente sensível ao contexto, é necessário que ele consiga identificar em que ponto de uma tarefa particular o usuário se encontra em um determinado instante. Logo, as arquiteturas de software que implementam o Modelo Semiótico deverão incluir, como parte do sistema de explicação, um agente de software que realize a **monitoração da execução**. Este agente empregará o(s) modelo(s) de tarefa que se encontra(m) armazenado(s) na *ADKB* para alimentar uma base de dados da execução — o **histórico de execução** —, deixando registrados os rastros das ações executadas pelo usuário durante sua seção de trabalho, sejam elas corretas ou não. Estes rastros é que possibilitarão ao agente de software do sistema de explicação identificar o contexto de trabalho do usuário. Além disso, eles também poderão ser empregados na implementação de um mecanismo de revogação[36].

Um dos requisitos identificados no Capítulo 3 para o software extensível é a necessidade de manutenção do princípio do Contínuo Semiótico entre a *EUPL*, a *UILx* (a parte extensível da *UIL*) e a *UEL*. Tal requisito é essencial para que os textos da *UEL* resolvam as dúvidas dos usuários satisfatoriamente. Uma forma segura de atingir este objetivo é fazer com que o texto da *UEL* seja gerado a partir das representações que se encontram na *ADKB*. Assim, o texto da *UEL* deverá ser gerado a partir das relações existentes entre as entidades que compõem os modelos do domínio e do software em resposta à pergunta do usuário levando em conta o seu contexto. No entanto, é importante salientar que nem todo o texto será gerado automaticamente. Deve existir um grão mínimo de explicação — arbitrariamente definido pelo designer do software —, que deve ser incorporado à *UEL* da aplicação na forma de texto fixo e que será responsável pelo último nível de explicação possível de ser apresentado ao usuário. Este mesmo tipo de texto deverá ser requisitado ao usuário durante o processo de criação de suas extensões de forma a manter este grão mínimo de explicação.

---

[36] *Undo.*

Deste modo, não apenas estaremos resolvendo o problema de manutenção do Contínuo Semiótico entre os três tipos de linguagens que são empregadas em uma aplicação extensível, como também resolveremos o problema da evolução da *UEL*, uma vez que as extensões realizadas pelo usuário deverão ser incorporadas ao software por meio da atualização da *ADKB* (conforme será descrito no Capítulo 6). Assim sendo, os textos gerados na *UEL* estarão sempre atualizados, podendo responder satisfatoriamente às dúvidas dos usuários, inclusive sobre as extensões por eles incluídas no software.

Os sistemas de tratamento de erros das arquiteturas de software atuais normalmente não levam em conta a diferença de conhecimento entre os programadores e os usuários finais. Assim, eles por vezes reportam um erro ocorrido em uma camada interna do software usando elementos do nível da linguagem de programação. Tais elementos são compreensíveis somente para as pessoas que detenham um bom conhecimento de programação. Como um usuário final normalmente não detém tal conhecimento, ele não conseguirá criar um modelo conceitual que lhe permita compreender a origem do erro. Uma solução para este problema, que é independente do tipo de arquitetura de software, está em produzir um texto explicativo que respeite a ontologia de domínio e de aplicação familiar aos usuários finais. Tal solução implica a alteração da forma de construção do software, de modo que os programadores devem traduzir o erro ocorrido em um determinado nível, através das diferentes camadas de software, até que ele atinja a camada de interface e seja reportado numa linguagem compreensível.

Além disso, freqüentemente, estes sistemas de tratamento de erros também desconsideram o contexto de ocorrência do erro, sendo este um problema até mais grave que o anterior. Este fato faz com que o mesmo texto explicativo seja reportado em situações bastante diferentes (algumas vezes até para erros diferentes). Como no caso anterior, o usuário também não conseguirá criar um modelo conceitual que lhe permita compreender a origem do erro, ou pior, ele poderá ser confundido em sua compreensão devido ao uso do mesmo texto em contextos totalmente divergentes. A única solução para este problema é empregar um mecanismo de tratamento de erros que considere o contexto de ocorrência do erro. Contudo, esta solução não é facilmente implementável nas arquiteturas atuais de software.

Assim, na arquitetura proposta no *SERG*, o **sistema de tratamento de erros** fará uso do histórico da execução criado pelo agente de software de monitoração da execução juntamente com uma descrição da tarefa atual, armazenada na *ADKB*, para identificar o contexto de ocorrência do

erro. Uma vez identificado este contexto, ele será passado para o agente de software gerador de textos do sistema de explicação que, assim, estará apto à geração de uma **mensagem explicativa situada**. Esta mensagem permitirá ao usuário compreender a origem de seu erro e, portanto, efetivar sua correção de forma a poder concluir sua tarefa. Deste modo, o sistema de tratamento de erros está ligado à **manutenção do princípio da Abstração Interpretativa** na medida em que ele isola as camadas inferiores do software da *UIL*.

O elemento primordial em uma arquitetura de um software extensível é o sistema de extensão. É possível dar suporte a vários dos paradigmas de programação, descritos no Capítulo 2, em uma mesma aplicação, como ocorre nas aplicações comerciais atuais. Todavia, dada a abordagem à tarefa de *EUP* adotada neste trabalho, é muito difícil introduzir o modelo de extensão aqui proposto em um software implementado com as arquiteturas tradicionais. Isto decorre da necessidade de **manutenção do princípio do Contínuo Semiótico** entre a *EUPL*, a *UILx* e a *UEL* do software que é um dos requisitos básicos deste modelo e que opera no momento em que as linguagens de representação e interação são projetadas. Se este projeto não contempla explicitamente esta meta, a verificação do Contínuo Semiótico é uma questão meramente probabilística.

Apesar de ser possível introduzir uma *ADKB* a um software já existente (criando, assim, uma representação comum entre os elementos da *UIL* e da nova *EUPL*), garantir a validade do princípio da Abstração Interpretativa para a *UIL* é uma tarefa que requer a re-implementação de todo o sistema de tratamento de erros e a introdução de um sistema de monitoração da execução, como descrevemos anteriormente. Se não bastasse esta dificuldade, também, será necessário re-implementar o sistema de explicação para garantir o Contínuo Semiótico com a *UEL*. Assim sendo, é possível dizer que uma aplicação deverá ser projetada desde seu início para ser extensível, e não ser modificada *a posteriori* para este fim [DE SOUZA '01].

É importante observar que neste trabalho estaremos interessados na realização de extensões produzidas pelo emprego de uma linguagem textual — a *EUPL*. Todavia, também é possível realizar extensões ao software por meio da sua própria *UIL*, empregando para isto alguns mecanismos de extensão não convencionais baseados em analogias e metáforas, conforme discutido em [BARBOSA '99]. É relevante salientar que a linguagem-tipo para *EUP* proposta neste trabalho também pode suportar a extensão pelos mesmos tipos de mecanismos baseados em analogias e metáforas, porém, por expressão textual. Para isto é necessário que sejam introduzidos

nos modelos armazenados na *ADKB* os mesmos tipos de relações entre entidades empregadas em [BARBOSA '99], devidamente adaptadas para a *KRL* da *ADKB*, e também os mecanismos de inferência lá definidos.

Assim, a tarefa do sistema de extensão será basicamente interpretar os textos da *EUPL*, com todos os seus mecanismos lingüísticos derivados da LN, realizando a manutenção da *ADKB* e garantindo a manutenção do Contínuo Semiótico entre a *EUPL*, a *UILx* e a *UEL*. Dada a complexidade desta tarefa, será necessário o emprego de um agente de software específico para realizar a interpretação de textos na *EUPL*. Este agente deverá ter conhecimento lingüístico que lhe possibilite a resolução de referências anafóricas e elípticas, a manutenção automática da classificação dos objetos presentes no texto da extensão, e a resolução de metáforas e metonímias* (caso estas sejam aceitas na instância da linguagem-tipo para *EUP* que está sendo processada). Ele será mais bem especificado na Seção 2.1 e, apesar do conhecimento lingüístico do qual é dotado, poderá ter de recorrer ao usuário nos momentos em que não conseguir resolver as referências na linguagem, obtendo assim uma interpretação aceitável para a extensão do usuário. Ele deverá, também, atuar juntamente com um grupo de agentes inteligentes de software específicos para a construção de interface que irão compor o **sistema de apoio à extensão**, para garantir a manutenção do Contínuo Semiótico entre a *EUPL*, a *UILx* e a *UEL*. Tais agentes auxiliarão o usuário na criação das interfaces necessárias às extensões por ele realizadas e serão mais bem especificados no Capítulo 6. De um modo geral, eles farão uso dos modelos do domínio e do software, principalmente das regras de design armazenadas na *ADKB*, para realizar a seleção dos *widgets* a serem empregados nos diálogos entre usuário e sistema, fazendo com que todo elemento de interação representado na *EUPL* tenha sua contraparte na *UIL*. Assim, serão necessários agentes de software específicos para cada uma das formas de entrada/saída disponibilizadas pela aplicação (ou seja, para cada um dos canais de comunicação disponíveis no software). Os agentes de software do sistema de extensão empregarão o sistema de tratamento de erros e de explicação para responder às dúvidas dos usuários e para tratar os erros que surgirem durante a realização da extensão.

A seguir, será discutida a estrutura da *EUPL*, e, na Seção 3 deste capítulo, será discutida a estrutura dos agentes de software e da *KRL* que deverão ser empregados na *ADKB*.

2. Uma linguagem-tipo para *EUP* para o Modelo Semiótico

A definição de uma linguagem de extensão para usuários finais está intimamente ligada ao modelo para a tarefa de *EUP* a ser empregado. Esta ligação advém do fato de ser esta linguagem o meio para o usuário expressar a realização das tarefas requeridas neste processo. A especificação de uma linguagem de programação sempre suscita questionamentos sobre quais mecanismos devem ou não estar presentes. Por exemplo, em [MYERS '92A], Dertouzos [DERTOUZOS '92] e Cordy [CORDY '92] fazem propostas bastante diferentes quanto à composição de uma linguagem de extensão, ambos apresentando justificativas bastante razoáveis para seus pontos de vista. Na mesma fonte, Myers, Smith e Horn [MYERS '92B] reportam conclusões de pesquisas na área de *EUP* que demonstram que uma linguagem de extensão deve ter os mesmos mecanismos básicos de qualquer linguagem de programação, sejam eles mecanismos para a declaração de objetos e de valores para estes objetos, para a seqüencialização e a iteração de ações, para a criação de condicionais e para a aplicação de funções e a criação de abstrações, possibilitando assim a criação de novos tipos e ações. A pergunta relevante então é: O que diferencia uma linguagem de extensão para usuários finais de uma linguagem de programação?

A diferença primordial entre as linguagens de programação e as linguagens de extensão para usuários finais reside no fato de que os usuários finais não apresentam conhecimento formal algum de programação, fato este nem sempre considerado nos trabalhos acima citados. Um efeito direto desta desconsideração é que os usuários finais necessitam explicitar muito mais detalhes para a realização de uma tarefa nas linguagens atuais de extensão do que eles estão acostumados a fazer no seu dia-a-dia. Em vista disso, apesar de estes trabalhos serem uma fonte muito útil de informação sobre os mecanismos que devem estar presentes em uma linguagem de extensão para programadores, o nível de linguagem por eles proposto não apresenta recursos comunicativos ricos o suficiente para que os usuários finais possam se expressar de forma natural, ou seja, sem que eles tenham que apelar para maiores conhecimentos sobre a estrutura do processador da linguagem utilizada. Os usuários finais necessitam de uma linguagem mais rica em recursos comunicativos que oculte as idiossincrasias das máquinas que vão processá-la, ajudando-os assim a superar a falta de conhecimento formal de programação.

Trabalhos mais recentes, como o de Pane *et al.* [PANE '01], também apontam para o fato de que a forma natural de expressão dos usuários finais é bem diferente da empregada nas linguagens de programação atuais. Seus resultados mostram, por exemplo, que os mecanismos de controle

mais comumente empregados pelos usuários finais são expressos no estilo de uma linguagem de eventos ou de regras de produção; que as interações sobre múltiplos objetos são comumente expressas implicitamente, através da operação sobre conjuntos de objetos; e que os usuários finais usam diferentes formas lingüísticas para criar conjuntos, como por exemplo, o uso de determinantes *all, each, every* e o uso do plural. Tais resultados são bastante significativos para a definição de linguagens de extensão para usuários finais, pois expressam a necessidade de mecanismos lingüísticos mais elaborados do que os presentes nas linguagens de programação atuais.

Na definição da linguagem-tipo para *EUP* para o Modelo Semiótico, fizemos estudos paralelos aos de Pane a partir de um ponto de vista distinto, reportados em [DA SILVA '97B] e no Capítulo 4 deste trabalho, que nos levaram a resultados que corroboram com os resultados por eles obtidos. Como foi observado no Capítulo 4, nestes estudos nos valemos de uma observação feita por Linden, em um artigo sobre design automático de software [LINDEN '91], em que ele nos alerta para o fato de que "programação e planejamento são atividades análogas". Também nos amparamos no fato apontado por Nardi de que as pessoas comuns estão aptas a usarem linguagens formais [NARDI '93] (como, por exemplo, a linguagem da matemática, a de representação de esquemas de bordados e a de anotação de escore e de movimentações em jogos). A partir destas observações, exploramos os tipos de planos que estas pessoas empregam diariamente e identificamos que elas estão acostumadas a ler e escrever receitas e instruções de faça-você-mesmo, e também a ler instruções de manuais de operação de máquinas, apesar de elas geralmente não escreverem este último tipo de texto. É interessante observar que todos estes textos são, na verdade, representações de planos — isto é, "representações-a-*posteriori*" que, normalmente, empregamos para apresentar uma explicação racional para nosso comportamento [SUCHMAN '87]. Isto revela uma conexão muito forte entre planos e explicações, e mostra que uma estrutura de plano é sempre uma estratégia representacional que usamos para transmitir nossas idéias aos outros. No caso de *EUP*, um plano será uma representação racional para um processador simbólico específico.

Nosso objetivo básico com esta abordagem foi tirar proveito dos ricos recursos de comunicação encontrados nestas linguagens de planos e, assim, atacar o ponto fraco para *EUP* das linguagens de programação e de extensão atuais. Deste modo, passamos a pensar em um procedimento para a definição de uma extensão ao software como um "plano elaborado por um usuário final". E, ao invés de pedirmos aos usuários finais que expressem seus planos para a

realização de alguma tarefa artificialmente elaborada, analisamos diretamente os planos expressos ou consultados por pessoas comuns no dia-a-dia, na forma de receitas e instruções de faça-você-mesmo.

Neste ponto, é importante salientar que a capacidade de ler um texto em uma determinada língua não implica diretamente a capacidade de escrever textos nesta mesma língua. Portanto, a capacidade de leitura é uma condição necessária, mas não suficiente, para a de redação. Sua posse, normalmente, possibilita à pessoa criar textos básicos numa língua. Contudo, quando se trata de um texto mais elaborado (isto é, um texto que apresenta uma estrutura de discurso mais formalizada), é muito comum que as pessoas que sabem ler este tipo de texto não detenham o conhecimento desta estrutura formal de discurso. Assim, nossa abordagem procurará tirar proveito da familiaridade dos usuários finais com este tipo de linguagem de planos para promover a aquisição da estrutura sintática da linguagem. Além disso, irá prover, como parte do Modelo Semiótico para a tarefa de *EUP*, um processo de criação de extensões que guiará o usuário na construção da estrutura formal de discurso correta para sua extensão.

Assim, buscamos, através de uma análise lingüística da linguagem empregada nas linguagens de planos das pessoas, identificar não somente a estrutura sintática das sentenças empregadas neste tipo de textos, mas, também, sua estrutura de discurso, ou seja, os aspectos comunicativos que tornam estes planos de fácil compreensão. Verificamos que estes textos são expressos em um subconjunto bastante restrito da linguagem natural, o subconjunto das sentenças imperativas. E que, apesar de este subconjunto ser sintaticamente limitado, ele é muito rico comparativamente ao que se encontra em linguagens de programação e, também, demonstra algumas dependências com o domínio de trabalho. Deste modo, determinamos as estruturas mais comumente empregadas neste subconjunto da linguagem natural e nos baseamos nelas para a definição de nossa *EUPL* esperando obter um balanceamento satisfatório entre o poder de expressão e a complexidade de *parsing* da linguagem. O *corpus* empregado nesta análise foi composto por receitas e instruções de faça-você-mesmo encontradas na *Internet*.

Apesar do *corpus* empregado nesta análise ser referente à língua inglesa, as observações aqui relatadas também são válidas para outras línguas, visto termos analisados elementos que pertencem aos mecanismos “lingüísticos universais”* [LYONS ’81] como, por exemplo, a estrutura frasal das diferentes formas de se referenciar um objeto em uma língua.

Dada nossa visão de uma extensão como um plano desenvolvido pelos usuários finais, uma possibilidade seria empregar como linguagem de extensão uma linguagem de planejamento já desenvolvida dentro da área de IA. Todavia, em [DA SILVA '97B] também apresentamos uma revisão destas linguagens que nos mostra não ser esta uma boa opção. Dentre as conclusões que obtivemos nesta análise, as principais são:

1. As linguagens de planejamento desenvolvidas pela comunidade de IA empregam Lógica de Primeira Ordem para expressar ações, objetivos e estados. Porém, os usuários finais não estão acostumados com a semântica dos conectivos empregados em tal Lógica, nem tampouco com a noção de implicação lógica.
2. A forma atual das linguagens de planejamento desenvolvidas pela comunidade de IA não favorece os objetivos de comunicação necessários a uma linguagem de extensão para usuários finais.

Deste modo, concluímos que o emprego de uma linguagem de planejamento derivada diretamente das linguagens desenvolvidas pela comunidade de IA não acarretaria bons resultados. E, portanto, a melhor opção será mesmo a definição de uma nova linguagem baseada em planejamento, mas que valorize os aspectos comunicativos necessários para facilitar a compreensão dos textos pelos usuários finais, tais como os mecanismos para manutenção da **coesão*** e **coerência textual*** dentro de um discurso.

2.1. A definição de uma linguagem-tipo para *EUP*

Em estudos preliminares [DA SILVA '97B], demonstramos que todos os mecanismos mencionados em [MYERS '92B] se encontram de uma forma ou de outra presentes nas linguagens de planos empregadas pelas pessoas no seu dia-a-dia. Assim, como resultado de nossa análise lingüística daquelas linguagens, apresentada no capítulo 4, partiremos do pressuposto de que uma *EUPL* cuja base seja estabelecida sobre um subconjunto bem demarcado da linguagem natural deverá ser de fácil aprendizagem pelos usuários finais, pois oferece recursos de comunicação mais poderosos, que já são conhecidos destes usuários.

Os resultados de nossos estudos também mostram que, apesar dos planos empregados pelas pessoas conterem uma seqüência de instruções, a semântica das ações neles descritas é semelhante à empregada nas linguagens de programação funcionais. Assim, a linguagem-tipo para *EUP* aqui proposta deverá empregar uma semântica funcional que focalize a transformação dos

objetos do domínio e evite o emprego direto de atribuição, pois as pessoas não estão acostumadas a este mecanismo. Ela deverá incluir ainda um mecanismo especial de referência a objetos, o qual permita o emprego de anáforas* e elipses*, de raciocínio classificatório e, possivelmente, de resolução de metáforas e metonímias [BARBOSA '99]. Deste modo, uma linguagem-tipo para *EUP* que apresente recursos comunicativos semelhantes àqueles descritos no subconjunto das sentenças imperativas da linguagem natural deverá conter pelo menos os mecanismos de: aplicação de operações, condicionais e iteradores e, também, deverá ter capacidades metalingüísticas para permitir a definição de novas abstrações. Tal conjunto de mecanismos juntamente com a semântica proposta deverá prover à *EUPL* seus recursos desejáveis de comunicação.

Um importante fato a ressaltar é que a linguagem aqui proposta é uma linguagem de extensão e, portanto, pressupõe a existência de algo a ser estendido (no presente caso, o software e a sua *ADKB*). Além disso, nossa abordagem impõe uma limitação às extensões possíveis: uma aplicação deve vir com um conjunto de recursos que não podem ser revogados ou destruídos pelo usuário, ou seja, no Modelo Semiótico as extensões possíveis de serem feitas sobre uma aplicação possuem **caráter monotônico**. Esta restrição preserva o significado mínimo da aplicação e nos mantém em consonância com a abordagem da Engenharia Semiótica, que vê o software como uma mensagem única e unidirecional do designer para os usuários.

2.1.1. Conceitos básicos de uma linguagem-tipo para EUP

A linguagem-tipo para *EUP* proposta neste trabalho é, assim como as LNs, uma linguagem multiparadigma. Ela gera uma linguagem eminentemente funcional com aspectos das linguagens orientadas a objetos, principalmente no que tange a manipulação das entidades do domínio. Todavia, seu **mecanismo de referenciação a objetos** é bastante diferente daquele empregado nas linguagens de programação atuais fazendo uso de mecanismos lingüísticos como, por exemplo, o conceito de contextualização e de foco de discurso* para aumentar a coesão textual e, assim, reduzir a carga cognitiva imposta aos usuários finais na interpretação dos programas nela escritos. Esta linguagem está dividida basicamente em duas partes: um **núcleo operativo** e uma **metalinguagem**. O núcleo operativo de uma *EUPL* instanciada da linguagem-tipo aqui proposta contém o mecanismo de referenciação a objetos da linguagem e os mecanismos de controle que podem ser empregados na descrição de ações. A metalinguagem contém mecanismos para a declaração de novas entidades ou extensão de antigas. Deste modo, este tipo de *EUPL* permitirá aos seus usuários a manipulação dos seguintes elementos:

- **Entidades** — que representam os elementos do domínio pertencentes ao modelo da aplicação que são visíveis e manipuláveis pelos usuários finais através da interface do software;
- **Atributos** — que representam características das entidades do domínio que são visíveis e manipuláveis pelos usuários através da interface do software;
- **Partes** — que representam entidades que compõem (fazem parte ou estão agregadas a) uma outra entidade do domínio e que são visíveis e manipuláveis pelos usuários finais através da interface do software;
- **Ações** — que representam as operações que uma entidade pode realizar no domínio e que tenham seus resultados visíveis para os usuários finais, ou seja, somente são consideradas as operações que provocam uma modificação direta na interface do software.
- **Interfaces** — que representam os mecanismos de ativação e os canais de comunicação que os usuários podem utilizar para ativar uma ação ou para comunicar a necessidade de dados e/ou os resultados da realização de ações; e
- **Relações** — que representam as inter-relações entre as diferentes entidades do domínio que são visíveis e manipuláveis pelos usuários finais através da interface do software.

A exigência de que todos os elementos sejam visíveis e manipuláveis pelos usuários finais procura deixar claro que o modelo da aplicação empregado pelos usuários finais será o **modelo mínimo** necessário para a compreensão do funcionamento do software, e não o modelo empregado pelo designer na construção do software que, normalmente, contém um número muito maior de elementos, necessários somente à compreensão da implementação do software em um ambiente operacional específico.

Neste ponto, é muito importante ressaltar que, assim como o *corpus* empregado na análise descrita no Capítulo 4 refere-se à língua inglesa, a linguagem-tipo para *EUP* aqui proposta também será descrita nesta língua. Esta escolha reflete a intenção de que este trabalho possa ser avaliado por uma comunidade mais ampla sendo, assim, justificada devido à possibilidade de publicar seus resultados para a comunidade internacional de informática. As propostas e os resultados deste trabalho valem para outras línguas, feitos os devidos ajustes nas estruturas lingüísticas peculiares da língua em questão. Uma gramática para uma *EUPL* na língua portuguesa, derivada da linguagem-

tipo para *EUP* aqui proposta, pode ser obtida, por exemplo, a partir do trabalho de Souza e Silva e Koch. [DE SOUZA E SILVA '96].

2.1.2. O núcleo da linguagem

O diferencial da linguagem proposta neste trabalho, em relação as demais linguagens de programação e extensão atuais, está no seu **mecanismo de referenciação**. Este mecanismo possibilita a descrição de referências a objetos empregando uma gramática reduzida da linguagem natural que permite ao usuário o emprego de DETERMINANTES como, por exemplo: *all*, *every*, *each*, *a* e *an*, que operam como **quantificadores** sobre os objetos de uma ação; ORDINAIS, como por exemplo, *next*, *first*, *last* e *previous*, que operam como **seletores** para o caminhamento em objetos estruturados; de ANÁFORAS, na forma de pronomes ou determinantes, como por exemplo: *it*, *them*, *their* e *its,* e *the*, *this* e *these*, que operam como **ponteiros** para objetos previamente referenciados do texto; de ADJETIVOS e NOMES PRÉ-MODIFICADORES* como, por exemplo: "*red*" *car* e *the message text*, e de SINTAGMAS* PREPOSICIONAIS como, por exemplo: "*...of the car*" ou "*...in the box*", que operam como **qualificadores** na especificação de um atributo ou parte de um objeto.

A gramática básica[37] da sub-linguagem de referenciação a objetos desta linguagem-tipo para *EUP* está descrita na Figura 19 abaixo e foi baseada em [BIBER '99]. É interessante observar que ela permite a especificação de um conjunto bastante rico de referências que não está disponível nas linguagens de programação atuais. É também importante ressaltar que esta gramática deve ser usada exclusivamente na interpretação de textos de uma instância da linguagem-tipo proposta, pois ela aceita mais sentenças do que as que são geralmente consideradas válidas em um texto na LN. As sentenças extras que ela aceita são consideradas incompletas do ponto de vista sintático, mas são comumente usadas no discurso diário dos usuários finais. Assim, este relaxamento na gramática proporciona um aumento na usabilidade da linguagem-tipo proposta, mas impossibilita a geração de texto a partir dela.

```
REFERE → SNOUNP [SEP_M REFERE]
SEP_M  → ',' | 'and'
SNOUNP → DETERM [PREMOD] HEADER [POSMOD]
HEADER → PPNAME | PRONOU | VARIAB | NUMBER
PPNAME → (A-Z, a_z, 0_9, -, _)* | STRING      % um nome próprio
STRING → '"' (caracteres válidos)* '"'
```

[37] Falamos em gramática básica pois ela não é completa, mas cobre o percentual mais significativo dos tipos de referências possíveis neste subconjunto da linguagem natural [BIBER '99].

```
PRONOU → 'it' | 'its' | 'them' | 'their'
VARIAB → '<' PPNAME '>'
DETERM → [PREDET] CENDET [GENITV] [POSDET]
PREDET → 'all' | 'any'
CENDET → ARTICL | 'each' | 'every' | 'this' | 'these' | ε38
ARTICL → 'a' | 'an' | 'the'
GENITV → PPNAME [''s'| ''']
POSDET → [ORDINA] [CARDIN]
ORDINA → 'first' | 'last' | 'previous' | 'next'
CARDIN → 'one' | 'two' | …
PREMOD → ADJECT | PPNAME
ADJECT → (A-Z, a_z, 0_9, -, _)*   % um adjetivo (o valor de um atributo)
POSMOD → PREP_P PREM_0 PNAME
PREP_P  → 'of' | 'in'
PREM_0  → ARTICL [POSDET]
```

Figura 19: Gramática da sub-linguagem de referenciação de objetos de uma *EUPL*.

A Tabela 1 apresenta uma amostra dos tipos de referências a objetos que podem ser gerados pela gramática da Figura 19 e que apresentam um mapeamento semântico válido numa *EUPL* instanciada da linguagem-tipo aqui proposta. Os exemplos presentes nesta tabela foram tirados de um domínio de aplicações que recebem e enviam *e-mails*.

Tabela 1: Tabela com exemplos dos tipos de referencias a objetos válidas para a gramática básica de uma *EUPL* instanciada da linguagem-tipo aqui proposta.

Tipo da Referência	**Exemplo**
"noun" (a proper name)	"personal"
<noun> (a variable)	<message>
[a/an] noun	[a] message
the/this noun	the message
the $noun_{ip}$ of [the] noun	the sender of the message
It	$_{get}$ it
its noun	its $_{sender}$
the/these noun(s)	these messages
the ordinal noun	the last message

[38] *Zero article*

(where ordinal = next/last/first/previous)	
the noun $noun_{ip}$ (only the partitive relation)	the message text
the adjective noun	the urgent message
[the] noun's $noun_{ip}$ (only the specifying genitive)	the message's sender
the ordinal noun $noun_{ip}$ (where ordinal = next/last/first/previous)	the first message attachment
the ordinal adjective noun (where ordinal = next/last/first/previous)	the first urgent message
Noun(s)	messages
all [the] noun(s)	all the messages
each/every noun	each message
the noun $noun_{ip}$(s)	the message attachments
the adjective noun(s)	the urgent messages
Them	send them
their noun(s)	their messages
all [the] $noun_1$(s) of/in the $noun_2$	all the messages in the mailbox
all [the] adjective noun(s)	all the urgent messages

Uma gramática de linguagem natural contém, além das regras sintáticas presentes na Figura 19, um conjunto de restrições de ocorrência mútua entre os elementos modificadores de uma referência e também entre a forma assumida pelo núcleo da referência (nome ou pronome) e estes modificadores [QUIRK '72]. No caso da linguagem-tipo para *EUP* proposta, estas regras podem ser empregadas em vários níveis. Contudo, visando obter uma linguagem mais tolerante aos erros gramaticais comumente cometidos pelos usuários finais, deverão ser empregadas somente as regras que proíbam ao usuário criar referências para as quais não é possível inferir um mapeamento semântico válido. Deste modo, o relaxamento no número de restrições de ocorrência mútua permite aos usuários se expressarem de forma mais livre e mais natural, apesar de apresentarem erros gramaticais.

A semântica da sub-linguagem de referência da linguagem-tipo para *EUP* será dada pelo mapeamento de uma referência sobre os elementos presentes na *ADKB*. Este mapeamento resultará em uma **semântica composicional** que gerará basicamente três classes de referências:

1. As que determinam uma única instância e que, para isto, operam sobre uma única entidade da *ADKB* como, por exemplo, *"urgent message"*, [*a/the/this*] *message*, *the message text*, *the message's sender*, *the sender of the message*, *it* e *its sender*;
2. As que determinam uma única instância mas que, para isto, necessitam operar sobre um conjunto de entidades da *ADKB* como, por exemplo, *the last message*, *the urgent message*, *the first message attachment* e *the first urgent message*; e
3. As que de terminam um conjunto de instâncias na *ADKB* como, por exemplo, [*the/these*] *messages*, *all the messages*, *each message*, *the urgent messages*, *send them*, *their messages*, *all the messages in the mailbox* e *all the urgent messages*.

Para definir o processo interpretativo que mapeia uma referência numa *EUPL* instanciada da linguagem-tipo aqui proposta sobre os elementos da *ADKB*, vamos considerar que os elementos modificadores do núcleo do sintagma nominal, isto é os quantificadores, os seletores e os qualificadores, que descrevem esta referência juntamente com os ponteiros e o número (singular ou plural) do núcleo da referê ncia respeitam uma relação de precedência, como a encontrada entre os operadores de uma linguagem de programação convencional. Assim, teremos a seguinte ordem crescente de precedência entre estes elementos:

- Quantificadores (*a*, *an*,[39] *all*, *any*, *each*, *every*, cardinais e número do nome núcleo da referência: singular e plural[40]).
- Seletores (ordinais).
- Qualificadores (adjetivos, nomes no caso* genitivo, nomes pré-modificadores e pós-modificadores).
- Ponteiros (*the*, pronomes pessoais: *it*, *its*, *their*, *them*, e pronomes demonstrativos: *this*, *these*).

---

[39] É importante ressaltar que o uso dos artigos indefinidos em lingüística é totalmente diferente do seu uso em lógica matemática.

[40] Usado sob o pronto de vista intencional.

Além disso, diferentemente das linguagens de programação, não adotaremos uma regra de associatividade para o caso da existência de elementos com a mesma precedência, caso típico dos quantificadores, cardinais e o número do núcleo da referência, pois, do ponto de vista gramatical, eles deverão ser mutuamente excludentes ou apresentar concordância, visto desempenharem a mesma função e, portanto, somente um será empregado na geração da expressão interpretadora da referência.

O processo de interpretação precisará consultar a *ADKB* para identificar as entidades que satisfazem a referência. A fim de evitar a fixação de um tipo de específico de *KRL*, empregamos um conjunto de funções auxiliares que usam a categoria especificada na referência, seus modificadores e os elementos do seu contexto de uso como filtros, criando assim uma linguagem intermediária entre a *EUPL* e a *KRL* da *ADKB*. Por conseguinte, a montagem da expressão interpretadora será realizada de forma composicional empregando as funções:

- *Any* — que será utilizada para indicar que qualquer entidade da categoria especificada é suficiente para satisfazer a referência. Ela é empregada nos quantificadores *a*, *an* e *any* e quando o núcleo da referência está no singular.
- *All* — que será utilizada para indicar que todas as entidades da categoria especificada são necessárias para satisfazer a referência. Ela é empregada nos quantificadores *all*, *each* e *every* e quando o núcleo da referência está no plural.
- *Part-of* — que será utilizada para indicar que o elemento que satisfaz a referência é uma parte de uma entidade da categoria especificada. Ela é empregada para os qualificadores formados pelo genitivo de um nome e pelos nomes pré-modificadores e pós-modificadores.
- *Attribute-of* — que será utilizada para indicar que o elemento que satisfaz a referência é um atributo de uma entidade da categoria especificada. Ela é empregada para os qualificadores formados pelo genitivo de um nome e pelos nomes pré-modificadores e nomes pós-modificadores
- *Attribute-value* — que será utilizada para indicar que a instância que satisfaz a referência será aquela que tiver um atributo ou parte de uma entidade da categoria especificada cujo valor seja igual ao especificado pelo adjetivo empregado. Ela é empregada para o qualificador formado por um adjetivo.

- *First, last, next* e *previous* — que serão utilizadas para indicar que a entidade que satisfaz a referência será resultante da seleção do elemento, indicado pelo nome do ordinal, que pertence ao conjunto ordenado de objetos formado por todas as entidades na *ADKB* da categoria especificada. Ela é empregada para os seletores formados pelos ordinais.
- *Set-of* — que será utilizada para indicar que as entidades que satisfazem a referência formam um conjunto ordenado da categoria especificada. Ela é empregada conjuntamente aos seletores formados pelos ordinais.
- *Resolve* — que será utilizada para indicar que a entidade que satisfaz a referência será um elemento previamente referenciado neste mesmo texto. Esta entidade deverá ser obtida através da resolução do elemento anafórico presente na referência atual, respeitadas as restrições de categoria impostas pela referência e pelo contexto. Ela é empregada para os pronomes pessoais *it*, *its*, *their* e *them*, para os pronomes demonstrativos *this* e *these* e para o determinante *the*.

A função auxiliar *resolve* engloba um mecanismo de resolução de anáforas e, portanto, requer maior atenção. É importante ressaltar que o conjunto de anáforas aceito numa *EUPL* derivada da linguagem-tipo para *EUP* aqui proposta é bem restrito e seus possíveis contextos de ocorrência são bem controlados. Estas características tornam possível ao agente de software do sistema de extensão empregar um modelo de resolução de anáforas pronominais baseado em *history lists* como o descrito por Allen em [ALLEN '95]. Neste modelo, uma *history list* será composta de entidades do discurso que aparecem no **co-texto**[*] no qual a referência está inserida (isto é, o contexto local, formado pela a sentença em que a anáfora ocorre, mais os contextos anteriores, formado pelas sentenças anteriores à sentença em que a anáfora ocorre que pertencem ao mesmo texto). O algoritmo de resolução de anáforas empregará estas listas, juntamente com um conjunto de restrições de co-ocorrência (e de categoria no caso da *EUPL*) para implementar o efeito de recentidade[41], que diz que o referente de uma anáfora deverá ser a entidade mais recentemente mencionada no texto que respeite todas as restrições impostas. Empregando este modelo, é possível também incorporar ao algoritmo o tratamento do efeito de focalização[42], que leva em conta a entidade que é o **foco do discurso** em uma determinada sentença, e, assim, produzir um algoritmo de melhor qualidade.

---

[41] *Recency effect*

[42] *Centering effect*

Algumas particularidades relevantes surgem no tratamento do elemento determinante *the*. Normalmente, este elemento refere-se a uma entidade que é unicamente determinada no texto, porém ele pode estar referenciando uma entidade do cenário geral ou específico do discurso como, por exemplo, no fragmento "*The computer is a machine that ...*", em que se fala do conjunto de todos os computadores existentes, e no fragmento "*The computer above my desk ...*" onde se fala de um computador específico. Ele pode ainda referenciar uma entidade do contexto global ou local [ALLEN '95]. Existe, além disso, uma importante distinção entre seu uso na forma existencial, na qual ele declara a existência de uma única entidade que satisfaz a referência como, por exemplo, no fragmento "*The car is a vehicle ...*"; e na referencial, na qual ele é usado para referenciar uma entidade previamente conhecida no contexto como, por exemplo, no fragmento "*Paint the car in red ...*" onde o carro referenciado é um carro que deve ter sido introduzido em uma sentença anterior no texto. E há ainda algumas situações lingüísticas em que este elemento é opcional como, por exemplo, na referência do fragmento "*all [the] messages ...*" [QUIRK '72]. Aliado a estes está o fato, sugerido em testes realizados com usuários finais (relatado no Anexo I deste trabalho), de que é muito comum a ocorrência de erros sintáticos nos quais os usuários omitem este determinante na descrição de uma referência inclusa na extensão como, por exemplo, no fragmento "*... Put [the] beeper in [the] bag ...*" retirado dos resultados dos testes apresentados no Anexo I.

Os fatos acima citados dificultam o tratamento deste tipo de referência. Deste modo, nas *EUPLs* instanciadas da linguagem-tipo aqui proposta, o principal uso para um sintagma nominal que contenha o determinante *the* será o referencial e, portanto, podemos empregar o mesmo algoritmo de resolução de anáforas pronominais para resolver estas referências. O uso deste sintagma na forma existencial será considerado para os casos de introdução no discurso de entidades que descrevem partes de uma entidade previamente referenciada. Assim, é possível adaptar o algoritmo de resolução de anáforas pronominais para que a *history list* contenha também as partes das entidades referenciadas no contexto, conforme descrito em [ALLEN '95], e então englobar também o tratamento do caso do *the* existencial.

Voltando ao problema de atribuir uma semântica às referências, o primeiro ponto a ser tratado é a presença de um nome "próprio" em uma referência. É importante salientar que, diferentemente da linguagem natural, para uma *EUPL* um nome "próprio" será todo elemento, simples ou composto, que não é posteriormente analisado pelo interpretador, ou seja, ele será aceito como um "símbolo definido pelo usuário", que será único dentro do sistema. Assim, o nome

"*personal*" será interpretado como o símbolo *personal* que será unicamente determinado dentro da aplicação.

A presença de uma referência genérica, ou seja, composta somente pelo nome núcleo do sintagma nominal, pode ocorrer em dois casos: 1) se o nome for singular como, por exemplo, em *message*, a expressão gerada será: (ANY MESSAGE), 2) Se ele for um nome plural como, por exemplo, em *messages*, a expressão gerada será: (ALL MESSAGE). É importante citar que aqui também se incluem os casos de erros sintáticos em que o usuário omitiu o determinante *a* ou *an*. Estes casos somente podem ser determinados pelo contexto da sentença, uma vez que podem ser confundidos com os casos de erro sintático em que o usuário omitiu o determinante *the*. Como a ausência destes elementos é um caso muito comum de erro sintático, é interessante que a linguagem a tolere, mas que realize uma interpretação semântica correta.

A presença de um quantificador ou de um cardinal na referência exige, conforme citado anteriormente, a existência de concordância entre este elemento e o nome, núcleo da referência que especifica a categoria da entidade referenciada. Por exemplo, em *all messages* existe concordância entre o número de elementos referenciados pelo quantificador *all* (que especifica um conjunto elementos) e do nome núcleo *messages* que está no plural e, portanto, referenciando um conjunto de mensagens. Nos casos em que há concordância, ou seja, não há erro gramatical, a expressão interpretadora gerada será: (ALL MESSAGE). Nos casos em que não houver concordância, terá ocorrido um erro gramatical, porém, como este tipo de erro é muito comum, é interessante que o texto gerado pelo usuário final seja aceito como válido. Apesar disso, deve ser sinalizada ao usuário a presença do erro como uma forma de auxílio à aprendizagem da gramática da *EUPL*. No entanto, é possível determinar diretamente sua semântica uma vez que a introdução do elemento quantificador é mais forte, do ponto de vista intencional, do que a ausência de concordância e, portanto, o quantificador deverá prevalecer e a expressão interpretadora gerada será a mesma.

A presença de um elemento seletor na referência, isto é um ordinal, implica a necessidade de disponibilizar uma função para realizar o caminhamento sobre uma coleção de objetos. Teremos, assim, um novo problema que será determinar uma ordem entre estes objetos. A solução mais simples é introduzir um contador de instâncias que será incrementado a cada nova instância gerada de uma entidade. Deste modo, será criada uma ordem parcial entre as instâncias, que poderá ser empregada na ordenação do caminhamento. Assim, para uma referência como, por exemplo, *the last message*, será gerada a expressão interpretadora: (LAST (SET-OF (RESOLVE THE MESSAGE))), onde a

função auxiliar *set-of* deverá devolver um conjunto ordenado de instâncias da categoria especificada pela determinação da referência *the messages*. É possível também utilizar um atributo como indexador das instâncias de uma entidade, porém neste caso será necessária a definição de uma sintaxe que permita especificar este indexador.

A presença de um adjetivo qualificador na referência exigirá uma consulta extra à *ADKB*, visto ser necessário determinar quais instâncias da categoria de entidade especificada pelo núcleo da referência têm um atributo cujo valor seja igual ao especificado pelo adjetivo. Assim, para uma referência como, por exemplo, *the urgent message* será gerada a expressão interpretadora: (ATTRIBUTE-VALUE (RESOLVE THE MESSAGE) URGENT), que verificará se a instância de *message* retornada pela função auxiliar *resolve* tem um atributo com o valor *urgent*.

Dentre as referências que empregam como qualificador um nome no caso genitivo, somente serão tratados os casos em que este nome atue como especificador. Assim, o núcleo da referência será sempre um atributo ou parte da categoria especificada pelo nome no genitivo como, por exemplo, em *the message's sender*. Em vista disso, também será necessária uma consulta extra à *ADKB* para determinar qual a propriedade o núcleo da referência exerce no contexto definido pelo qualificador. Dependendo do resultado desta consulta, a expressão interpretadora gerada poderá ser: (ATTRIBUTE-OF (RESOLVE THE MESSAGE) SENDER) ou (PART-OF (RESOLVE THE MESSAGE) SENDER).

Dentre as referências que empregam como qualificador um nome pré-modificador ou um sintagma preposicional pós-modificador, somente serão tratados os casos que descrevam uma relação todo-parte entre o qualificador e o núcleo. Nesta relação, o núcleo da referência será um atributo ou parte da categoria especificada pelo nome pré-modificador ou pelo sintagma preposicional pós-modificador como, por exemplo, em *the message text* e *the text of the message* respectivamente. Assim, também será necessária uma consulta extra à *ADKB* para determinar a função que o núcleo da referência exerce no contexto definido pelo qualificador e a expressão interpretadora gerada obedecerá os mesmos casos previstos para o tratamento de nomes no genitivo, apesar da ordem sintática do elemento qualificador e qualificado estar invertida no caso do sintagma preposicional pós-modificador.

A presença, na referência, de um elemento que atua como um ponteiro poderá ocorrer em duas situações bem distintas: 1) quando o ponteiro é o núcleo da referência como, por exemplo, em *its name* ou *their*, e 2) quando o ponteiro é um determinante (*the*, *this* e *these*) como, por exemplo, em *this message*. No primeiro caso, será necessário resolver o elemento anafórico circunscrito por um

conjunto de restrições sobre a categoria válida de instância. Tais restrições são definidas em função do contexto em que a referência se encontra. No segundo caso, a categoria já está especificada pelo núcleo da referência. No entanto, é necessário que a instância tenha sido previamente referenciada no mesmo texto e, para o caso do determinante *the*, é necessário fixar o número de instâncias que serão devolvidas como resposta em função do nome núcleo da referência. Deste modo, as expressões interpretadoras geradas para os exemplos apresentados serão: (ATTRIBUTE-OF (RESOLVE ITS CONTEXT) NAME), (RESOLVE THEIR CONTEXT) e (RESOLVE THIS MESSAGE).

É importante observar que os quantificadores, os seletores, os qualificadores e os ponteiros (as anáforas) somente podem ser tratados corretamente em um nível mais alto da gramática da *EUPL*, ou seja, no nível do discurso. Somente neste nível é possível ao agente interpretador determinar o co-texto de ocorrência de uma referência e, assim, através do acesso à *ADKB* resolvê-la para as instâncias específicas do modelo da aplicação. A Tabela 2 apresenta exemplos das expressões interpretadoras geradas, para os tipos de referência apresentados na Tabela 2, juntamente com uma explicação de seu resultado.

Tabela 2: Tabela com exemplos de mapeamento semântico das referências de uma *EUPL* instanciada da linguagem-tipo aqui proposta sobre elementos da *ADKB*.

Tipo da Referência	Mapeamento
"noun" (a proper name)	NOUN O próprio nome '*noun*'.
<noun> (a variable)	(VAR NOUN)
[a/an] noun any noun	(ANY NOUN) Qualquer instância do tipo *noun* presente na *ADKB*.
the noun $noun_{ip}$ the noun's $noun_{ip}$ the $noun_{ip}$ of [the] noun	(ATTRIBUTE-OF (RESOLVE THE NOUN) $NOUN_{IP}$) ou (PART-OF (RESOLVE THE NOUN) $NOUN_{IP}$) Um elemento do tipo *$noun_{ip}$*, que é um atributo ou parte de uma instância do tipo *noun* presente na *ADKB*.
the/this noun	(RESOLVE THE/THIS NOUN) Uma instância do tipo *noun* presente na *ADKB* previamente referenciado no texto.
It	(RESOLVE IT CONTEXT) Uma instância presente na ADKB decorrente da resolução da anáfora it dentro do contexto de uso da referência.
its $noun_{ip}$	(ATTRIBUTE-OF (RESOLVE ITS CONTEXT) $NOUN_{IP}$) ou

	(PART-OF (RESOLVE ITS CONTEXT) NOUN$_{IP}$) Um elemento do tipo *noun$_{ip}$*, que é um atributo ou uma parte de uma instância decorrente da resolução da anáfora its dentro do contexto de uso da referência.
the adjective noun	(ATTRIBUTE-VALUE (RESOLVE THE NOUN) ADJECTIVE) Uma instância do tipo *noun* presente na *ADKB* que tem um atributo cujo valor é *adjective*.
the ordinal noun (where ordinal = next/last/first/previous)	(ORDINAL (SET-OF (RESOLVE THE NOUN))) A (primeira, última, próxima, ou anterior) instância do conjunto ordenado de todas as instâncias do tipo *noun* presentes na *ADKB*.
the ordinal noun noun$_{ip}$ (where ordinal = next/last/first/previous)	(ORDINAL (ATTRIBUTE-OF (SET-OF (RESOLVE THE NOUN)) NOUN$_{IP}$)) ou (ORDINAL (PART-OF (SET-OF (RESOLVE THE NOUN)) NOUN$_{IP}$)) O elemento do tipo *noun$_{ip}$* da (próximo, primeira, última ou anterior) instância do conjunto ordenado de todas as instâncias do tipo *noun* presente na *ADKB*.
the ordinal adjective noun (where ordinal = next/last/first/previous)	(ORDINAL (ATTRIBUTE-VALUE (SET-OF (RESOLVE THE NOUN)) ADJECTIVE)) A (próxima, primeira, última ou anterior) instância do conjunto ordenado de todas as instâncias do tipo *noun* presente na ADKB que tem um atributo cujo valor é *adjective*.
the/these noun(s)	(RESOLVE THE/THESE NOUN) O conjunto de todas as instâncias do tipo *noun* presentes na *ADKB* previamente referenciados no texto.
noun(s) all [the] noun(s) each/every noun	(ALL (RESOLVE THE NOUN)) O conjunto de todas as instâncias do tipo *noun* presentes na *ADKB* previamente referenciadas no texto.
the noun noun$_{ip}$(s) the noun(s)' noun$_{ip}$ all [the] noun$_{ip}$(s) of/in the noun	(ALL (ATTRIBUTE-OF (RESOLVE THE NOUN) NOUN$_{IP}$)) ou (ALL (PART-OF (RESOLVE THE NOUN) NOUN$_{IP}$)) O conjunto de todos os elementos do tipo *noun$_{ip}$* que são um atributo ou uma parte de uma instância do tipo *noun* presente na *ADKB*.
the adjective noun(s) all [the] adjective noun(s) each/every adjective noun	(ALL (ATTRIBUTE-VALUE (RESOLVE THE NOUN) ADJECTIVE)) O conjunto de instâncias do tipo *noun* presentes na *ADKB* que tenham um atributo cujo valor é *adjective*.
Them	(RESOLVE THEM CONTEXT) O conjunto de todas as instâncias de um mesmo tipo presentes na *ADKB* decorrentes da resolução da anáfora *them* dentro do contexto de uso da referência.
their noun$_{ip}$(s)	(ALL (ATTRIBUTE-OF (RESOLVE THEIR CONTEXT) NOUN$_{IP}$)) ou (ALL (PART-OF (RESOLVE THEIR CONTEXT) NOUN$_{IP}$)) O conjunto de todos os elementos do tipo *noun$_{ip}$* que são um atributo ou parte das instâncias decorrentes da resolução da anáfora *their* dentro do

	contexto de uso da referência.

É interessante observar que as expressões interpretadoras que envolvem funções auxiliares do tipo ***all*** representam um mapeamento sobre um conjunto de entidades da *ADKB* e, portanto, modelam iterações implícitas sobre as ações que empregam parâmetros que as contém. Para que elas possam ser corretamente combinadas com as expressões interpretadoras destas ações, é necessário que sejam re-escritas do formato original (APPLY ACTION (ALL (ELEMENT-SPECIFICATION))) para um formato que gere uma interação explícita (ITERATE (APPLY ACTION (ELEMENT-SPECIFICATION)) LAST). A função *iterate* é uma função de alta-ordem, com semântica semelhante às funções de mapeamento encontradas nas linguagens aplicativas, e *last* indicará o tipo do teste de término da iteração. Assim, por exemplo, para a ação especificada pela sentença *send all the messages*, deverá ser gerada a seguinte expressão interpretadora: (ITERATE (APPLY SEND (RESOLVE THE MESSAGE)) LAST).

Esta discussão encerra a parte descritiva do mecanismo de referenciação da linguagem-tipo para *EUP*. A parte do núcleo de uma *EUPL* que trata dos mecanismos de controle contém regras que possibilitam a definição de ações condicionais, de iteradores explícitos e a aplicação de ações sobre referências. Sua gramática pode ser visualizada na Figura 20.

```
ACTIONS → ACTION
ACTIONS → ACTION [END_M | SEP_M] ACTIONS
SEP_M   → ',' | 'and'
END_M   → '.'
ACTION  → COND_M LOGEXP SEP_M ACTIONS
COND_M  → 'if' | 'when'
ACTION  → NCON_M SEP_M ACTIONS
NCON_M  → 'if not'
EXPRES  → MATEXP | LOGEXP
MATEXP  → % uma gramática de expressões matemáticas válida no domínio
LOGEXP  → RELAM0 REFERE43
RELAM0  → 'there is'
LOGEXP  → REFERE0 [RELAM1 REFERE1 | RELAM2]
RELAM1  → 'belongs to' | 'is'
RELAM2  → 'is accepted'
ACTION  → ITERM1 ACTIONS PREPD LOGEXP
ITERM1  → 'repeat'
ACTION  → ITERM2 REFERE SEPM ACTIONS
ITERM2  → 'for'
```

[43] O não-terminal `REFERE` é descrito na gramática da Figura 19.

```
ACTION  → ITERM3 REFERE0 PREPP REFERE1 ACTIONS
ITERM3  → 'from'
ACTION  → BNAME PREP REFERE [ADJUNC1 [[SEPM] ADJUNC2 [[SEPM] ADJUNC3]]
ADJUNC  → (PREPG | PREPI | PREPD | PREPP) REFERE
PREP    → 'at' | 'on' | 'in' | 'up' | 'with' | 'for'
PREPP   → 'at' | 'on' | 'to' | 'in'
PREPI   → 'using' | 'with' | 'in' | 'on'
PREPG   → 'for' | 'to'
PREPD   → 'until' | 'up to' | 'while'
BNAME   → (A-Z, a_z, 0_9, -, _)*            % um verbo
```

Figura 20: Gramática que descreve os mecanismos de controle de uma *EUPL*

É relevante observar que o mecanismo de controle para a descrição de condicionais opera com uma semântica semelhante a das regras de produção encontradas na área de IA. A especificação da expressão lógica que funciona de guarda para a execução das ações poderá conter, além dos testes comumente empregados na matemática, testes específicos para o domínio da aplicação de forma a verificar condições específicas do sistema como, por exemplo, *belongs to*, que verifica se uma determinada instância pertence a uma entidade, *there is*, que verifica a existência de determinada condição, e *is accepted* que, verifica se o usuário aceitou determinada condição. O mecanismo aceita ainda a especificação de condições negativas como, por exemplo, *if not* que dispara as ações sempre que ocorrer a falha de um teste anterior (ou seja, ela faz parte do escopo do teste anterior).

Os iteradores explícitos somente terão uso prático quando se desejar repetir um conjunto de ações sobre um determinado objeto, pois, caso contrário, é mais natural o uso de iteradores implícitos. Os iteradores explícitos são de três tipos:

1. Os iteradores do tipo *repeat*, que testam uma condição de repetição após realizar um conjunto de tarefas;
2. Os iteradores sobre estruturas, que operam como os iteradores implícitos, porém realizam um conjunto de ações sobre os elementos da estrutura em vez de uma única ação; e
3. Os iteradores que operam sobre um intervalo de elementos permitindo, assim, limitar o número de elementos da estrutura que serão afetados pelo conjunto de ações.

A aplicação de funções sobre referências funciona com uma semântica funcional, porém, ela pode conter iteradores implícitos em seus parâmetros, caso em que terá de ser re-escrita de

modo a expressar uma iteração explícita, conforme citado anteriormente. Além disso, sua sintaxe é ligeiramente diferente da sintaxe dos mecanismos semelhantes empregados nas linguagens de programação convencionais, empregando as estruturas encontradas no subconjunto das sentenças imperativas da linguagem natural, o que a torna mais compreensível aos usuários finais. Deste modo, uma ação numa *EUPL* instanciada da linguagem-tipo para *EUP* aqui proposta pode operar sobre até quatro conjuntos de variáveis, que são descritos pelo objeto da ação e pelos adjuntos que compõem os casos de um verbo na linguagem natural como, por exemplo, em "*send* ***the*** *<message>* ***to all*** *contacts* ***of the*** *<group>.*".

O último elemento desta parte da linguagem é composto por uma sub-linguagem de expressões empregadas para a especificação de operações matemáticas e lógicas. Como dissemos anteriormente, a sub-linguagem de expressões lógicas será dependente do domínio. Porém, um ponto relevante a ser considerado é que, na introdução das operações relacionais deverá ser observado o fato de que os usuários finais interpretam de forma bastante diferente os operadores *and* e *or*, o que normalmente acarreta alguns problemas de interpretação na leitura do texto da *EUPL*. Também é relevante dizer que a sub-linguagem de expressões matemáticas deverá prover, além das formas matemáticas, também as formas lingüísticas dos operadores por ela empregados como, por exemplo, *plus*, *minus*, *times* e *divided by*, pois os usuários finais estão mais acostumados a seu uso. Por outro lado, é necessário alertar para o fato de que a linguagem-tipo para *EUP* proposta apresenta uma semântica funcional e, portanto, a maioria das transformações de valores será realizada pela aplicação de funções que, normalmente, serão primitivas[44]. Assim, a sub-linguagem de expressões matemáticas também apresentará alguma dependência com o domínio. Deste modo, não especificaremos estas sub-linguagens neste trabalho e somente apresentamos na gramática alguns exemplos úteis de expressões lógicas.

A semântica desta parte da linguagem será baseada nas linguagens de planejamento de IA como, por exemplo, a linguagem *ACT* proposta por Wilkins *et al.* [WILKINS '95]. A escolha deste tipo de mapeamento se deve à facilidade de geração de texto a partir de tais formalismos, conforme demonstram Appelt [APPELT '85], Hovy [HOVY '88] e Moore [MOORE '95]. Deste modo, a exata linguagem de planejamento a ser empregada irá variar segundo as necessidades de geração de texto da *UEL* que a aplicação deverá suportar. Deste modo, neste trabalho não iremos propor um mapeamento único para esta parte da gramática. Em vez disto, descreveremos um mapeamento

---

[44] As funções primitivas são funções que fazem parte da funcionalidade básica da aplicação.

possível para as regras da gramática proposta e mostraremos alguns exemplos com os seus respectivos mapeamentos.

Deste modo, as regras:

```
ACTIONS → ACTION [END_M | SEP_M] ACTIONS
ACTIONS → ACTION
```

definem uma seqüência de ações e serão mapeadas para uma seqüência de instruções e uma única instrução, respectivamente, resultando na geração das expressões:

```
((interpret ACTION)
 (interpret ACTIONS))
```

ou, no caso de uma única ação da expressão:

```
(interpret ACTION)
```

onde a função `interpret` chama recursivamente o interpretador para continuar a interpretação das ações internas. Como exemplos de uso destas regras temos:

... send ***the*** *message* ***to the*** *group ...*

(interpret "*send the message to the group*")

... create ***a*** *contact* ***and*** *add* ***it to the*** *group ...*

((interpret "*create a contact*"
 (interpret "*add it to the group*"))

As regras:

```
ACTION  → COND_M LOGEXP SEP_M ACTIONS
ACTION  → NCON_M SEP_M ACTIONS
```

definem ações condicionais que deverão fazer uso de um *flag* interno do sistema, ativado pelas expressões lógicas, para determinar se as ações especificadas após a condição serão ou não executadas. Para testar este *flag* são empregadas as funções auxiliares `on-true` e `on-false`. Logo, para a primeira regra deverá ser gerada a expressão:

```
((interpret LOGEXP)
 (on-true (interpret ACTIONS)))
```

e para a segunda a expressão:

```
(on-false (interpret ACTIONS))
```

como exemplos de uso destas regras temos:

… ***if the*** *group* ***belongs to the*** *address-book****,*** *send* ***the*** *…*

`((interpret` *"the group belongs to the address-book"*`)`

`(on-true (interpret` *"send the …"*`)`

… ***if not,*** *show* ***the*** *message "…" using* ***the*** *…*

`(on-false (interpret` *"show the message "…" using the …"*`))`

A seguir apresentaremos alguns exemplos de regras para especificação de expressões lógicas. Estas regras são as responsáveis por ativarem o *flag* interno do sistema que será testado pelas funções *`on-true`* e *`on-false`*. Seu mapeamento dependerá do domínio e exigirá a definição de funções auxiliares específicas. Deste modo, podemos ter, por exemplo:

LOGEXP → $RELA_{M0}$ REFERE
$RELA_{M0}$ → '**there is**'

que mapeia para:

`(exist? REFERE)`

LOGEXP → $REFERE_0$ [$RELA_{M1}$ $REFERE_1$ | $RELA_{M2}$]
$RELA_{M1}$ → '**belongs to**' | '**is**'
$RELA_{M2}$ → '**is accepted**'

que mapeia para:

(belongs-to? $REFERE_0$ $REFERE_1$)

ou:

(is? $REFERE_0$ $REFERE_1$)

ou:

(is-accepted? $REFERE_0$)

como exemplos de uso destas regras temos:

… ***when there is an*** *error …*

`… (exist? error) …`

If the *<group>* ***belongs to the*** *<address-book>…*

`…(belongs-to? (var group) (var address-book))…`

If the *<creation-of-a-new-group>* ***is accepted,*** *…*

`…(is-accepted? (var creation-of-a-new-group)) …`

As regras que definem os iteradores explícitos são mais complexas e requerem a definição de um novo escopo que marque a região que será repetida a cada iteração. No entanto, seu mapeamento pode ser semelhante ao realizado nas linguagens convencionais. Assim temos:

```
ACTION → ITER_M1 ACTIONS PREP_D LOGEXP
```

onde existe uma condição explícita (LOGEXP) para ser testada e a continuidade ou não da iteração será determinada por ela. Deste modo, será gerado o conjunto de expressões:

```
(loop
 (interpret ACTIONS)
 ((interpret LOGEXP)
  (on-true  exit)
  (on-false loop))
 )
```

onde a função *loop* marca o escopo da iteração e a função auxiliar *exit* encarrega-se de realizar a saída do processo de iteração. O retorno ao início do processo de iteração é marcado pela chamada recursiva da função *loop*. Como exemplo de uso desta regra temos:

Repeat
if the *colour* **of** *<object>* **is** *<colour>*, *go to* **its** *places* **and** *pick* **it**
up to the last *<colour object>* **of the** *world …*

```
(loop
 (interpret "if the colour of <object> is <colour>, go to its places and pick it")
 ((interpret "last <colour object> of the world")
  (on-true  exit)
  (on-false loop))
 )
```

A regra:

```
ACTION → ITER_M2 REFERE SEP_M ACTIONS
```

descreve o caso em que a iteração será determinada pela existência ou não de um novo elemento a ser iterado numa estrutura, o que será determinado pelo resultado das funções *get-first* e *get-next*. Assim, deverá ser gerado o conjunto de expressões:

```
((let element (get-first (determine REFERE)))
 (loop
  ((exists? element)
   (on-true (interpret ACTIONS))
   (on-false exit))
  (loop (get-next (determine REFERE)))
))
```

onde a função `let` introduz no escopo uma nova variável que referenciará o elemento que está sendo iterado e a função `determine` é responsável por determinar o conjunto de elementos que serão iterados a partir da referência especificada. Como exemplo de uso desta regra temos:

For all** <type> appointments **of the** agenda**,** set **its** day's color **to** <color>**.

```
((let element (get-first (determine "all <type> appointments of the agenda")))
 (loop
  ((exists? element)
   (on-true (interpret "set its day's color to <color>"))
   (on-false exit))
  (loop (get-next (determine "all <type> appointments of the agenda")))
))
```

A regra:

```
ACTION  → ITER_M3 REFERE_0 PREP_P REFERE_1 ACTIONS
```

descreve o caso em que a iteração também será determinada pela existência ou não de um novo elemento a ser iterado na estrutura, porém, neste caso, o número de elementos poderá ser limitado. Deste modo, será gerado o conjunto de expressões:

```
((let set (make-set (determine REFERE_0) (determine REFERE_1)))
 ((let element (get-first set))
  (loop
   ((exists? element)
    (on-true (interpret ACTIONS))
    (on-false exit))
   (loop (get-next set))
)))
```

como exemplo de uso desta regra temos:

From the** <first-element> mail **to the** <last-element> mail**,** change **their** status **to** 'Read'**.

```
((let set (make-set (determine "the <first-element> mail")
                    (determine "the <last-element> mail")))
 ((let element (get-first set))
  (loop
   ((exists? element)
    (on-true (interpret "change their status to 'Read' "))
    (on-false exit))
   (loop (get-next set))
)))
```

As regras:

```
ACTION → BNAME PREP REFERE [ADJUNC1 [[SEPM] ADJUNC2 [[SEPM] ADJUNC3]]
ADJUNC → (PREPG | PREPI | PREPD | PREPP) REFERE
```

definem a aplicação de uma ação a um conjunto de referências (parâmetros). É importante salientar que estas regras definem a única instrução que realmente produz alguma alteração efetiva no estado da aplicação. Seu mapeamento faz uso da função auxiliar *apply* e resultará respectivamente nas expressões:

```
(apply (interpret BNAME) (interpret REFERE)
       (interpret ADJUNC1)(interpret ADJUNC2)(interpret ADJUNC3))
```

e:

```
("PREP" REFERE)
```

Como exemplo de uso destas regras temos:

... *send* ***the*** *message* ***to the*** *contact* ...

(apply (send (interpret "*the message*") (interpret "*to the contact*")))

É importante ressaltar que os mecanismos de referenciação de uma *EUPL* instanciada da linguagem-tipo aqui proposta ajudam a eliminar muitos dos passos intermediários necessários à especificação de iterações. Além disso, o uso de seletores no caminhamento por estruturas torna implícita a complexidade inerente neste tipo de processamento possibilitando, assim, a eliminação de referências intermediárias quando se deseja falar de um elemento interno dentro de uma instância com estrutura complexa. Estas capacidades simplificam muito a descrição de iterações sobre estruturas complexas nestas *EUPLs*, possibilitando a especificação deste tipo de ação em um único passo, o que diminui a carga cognitiva sobre o usuário final na interpretação do texto da extensão.

Com a especificação da semântica deste conjunto de mecanismos concluímos a descrição do núcleo da linguagem-tipo para *EUP* proposta. Deste modo, as linguagens de extensão instanciadas desta linguagem-tipo passam a ter o mesmo poder computacional que uma linguagem de programação convencional. O que as distinguirá sensivelmente destas últimas serão seus recursos para referenciação a objetos, que ampliam muito seu poder expressivo, e seus recursos para a definição da interação com o usuário, que serão descritos na seção da metalinguagem a seguir.

2.1.3. A metalinguagem

A metalinguagem é a parte de uma *EUPL* responsável pela extensão da *ADKB*. Ela irá permitir a especificação de novas entidades na base de conhecimento e de novas propriedades para estas entidades e para as que já estão presentes na *ADKB*.

Antes de discutirmos a sintaxe e a semântica desta parte da linguagem-tipo para *EUP* aqui proposta, é interessante entendermos a estrutura da base de conhecimento que a apoiará. Primeiramente, é importante dizer que a *ADKB* será baseada em um sistema de *frames* [MINSKY '75]. A escolha por um sistema de *frames* se justifica pela facilidade de representação tanto de conhecimento declarativo quanto procedimental [WINOGRAD '75], facilitando, assim, a representação das ações. A estrutura representacional da *ADKB* será mista, considerando conceitos propostos por Brachman em [BRACHMAN '77], por Schank e Abelson em [SCHANK '78] e por Sowa em [SOWA '91]. Deste modo, sua estrutura representacional procura contemplar a representação de conhecimento declarativo e procedimental de forma a facilitar a geração de texto explicativo pela *UEL* quando for necessário. Assim, a *ADKB* será composta de um conjunto de entidades $\mathbf{E} = \{e_j \mid j>0\}$ no qual cada entidade $\mathbf{E_j}$ deverá ter, no mínimo, um conjunto de atributos $\mathbf{A} = \{a_k \mid k>0\}$; um conjunto de ações $\mathbf{B} = \{b_k \mid k>0\}$ e um conjunto de interfaces $\mathbf{I} = \{i_k \mid k>0\}$ e, opcionalmente, poderá ter um conjunto de partes $\mathbf{P} = \{p_k \mid k=0\}$ e um conjunto de relações $\mathbf{R} = \{r_k \mid k=0\}$. Na *ADKB*, toda entidade $\mathbf{E_i}$ deriva de outra entidade $\mathbf{E_j}$, isto é, apresenta uma relação estrutural *is-a* com outra entidade da *ADKB*, semelhante ao que ocorre nos modelos de orientação a objetos baseados em prototipação. A escolha por um sistema baseado em prototipação se justifica pelas descobertas de Rosch [ROSCH '78] de que a estrutura interna das categorias empregadas pelas pessoas parecem ser mais bem caracterizadas por uma estrutura de "tipicalidade" ao invés de condições necessárias e suficientes. Isto exige que exista na *ADKB* um conjunto de entidades predefinidas que dará suporte à estruturação do modelo, este conjunto é dependente do domínio e, segundo Bloom [BLOOM '98], será baseado nas intenções do designer ou do usuário. Além desta relação, é possível definir que uma entidade contém partes através da relação estrutural *part-of*. É importante ressaltar que toda parte é uma entidade definida na *ADKB* e que ela terá o mesmo tempo de vida que a entidade que a contém[45].

A taxonomia, formada pela relação estrutural *is-a* e sua inversa *subclasses*, é importante porque valida a inferência de propriedades e de inclusão de classes, permitindo, assim, a definição

[45] A menos que ela seja associada a outra entidade antes da destruição da entidade original que a contém.

de novas entidades empregando os princípios de diferenciação e complementação, de modo que somente seja necessário descrever as partes de uma entidade que a diferenciam de outra, reduzindo, assim, a quantidade de texto necessária à criação de uma extensão e facilitando a compreensão da extensão pelos usuários finais. Já a partonomia, formada pela relação estrutural *part-of* e sua inversa *parts*, não permite a inferência de propriedades [MARK '99], entretanto, ela é muito importante para a resolução de anáforas e elipses dentro do mecanismo de determinação de referências de uma *EUPL* instanciada da linguagem-tipo aqui proposta. Por último, uma entidade poderá ser composta ou não, ou seja, ela poderá representar a agregação de uma coleção de entidades, atuando como um agregado do tipo *set*, *list* ou *sequence*. Deste modo, a estrutura das entidades da *ADKB* terá a seguinte forma:

```
EntityName
       is-a           value          [enti1, enti2, …, entiN]
       subclasses     value          (nil/[enti1, enti2, …, entiN])
                      constraints    {entii ∈ EntityWorkingSet}46
                                     {∀i,j entii ≠ entij}
       attributes     value          (nil/[attr1, attr2, …, attrN])
                      constraints    {∀i,j attri ≠ attrj}
       actions        value          (nil/[acti1, acti2, …, actiN])
       interfaces     value          [inte1, inte2, …, inteN]
                      constraints    {∀i,j intei ≠ intej}
       part-of        value          (nil/[enti1, enti2, …, entiN])
       parts          value          (nil/[enti1, enti2, …, entiN])
                      constraints    {conci ∈ EntityWorkingSet }
                                     {∀i,j entii ≠ entij}
       compound       value          Boolean
                      element-type   (nil/entity)
                      constraints    {entity ∈ EntityWorkingSet }
```

Figura 21: Descrição da estrutura de uma entidade na *ADKB*.

Na *ADKB*, cada atributo $\mathbf{A_i}$ será igualmente representado como um elemento independente. Assim, será possível definir para cada um deles uma faixa de valores válidos (através da enumeração destes valores ou da definição de uma categoria para o atributo) e seu valor *default*. O valor atual de um atributo poderá ser atribuído: 1) pelo usuário, 2) através das mudanças na aplicação, ou 3) ser computado pelo sistema (caso em que será necessário definir um conjunto de instruções para seu cálculo).Em vista disso, a estrutura de um atributo na *ADKB* terá a seguinte forma:

---

[46] O ***EntityWorkingSet*** é o conjunto de entidades presentes na *ADKB* em um determinado momento.

```
EntityName-AttributeName
        value-range     value           (enumeration/entity)
                        constraints     {entity ∈ EntityWorkingSet}
        default         value           (a value)
                        constraints     {value ∈ value-range}
        computed        value           Boolean
                        instructions    (nil/[inst_1, inst_2, …, inst_N])
        value           value           (a value)
                        constraints     {value ∈ value-range}
                                        {computed = true → value = nil}
```

Figura 22: Descrição da estrutura de um atributo na *ADKB*.

Na *ADKB*, as interfaces $\mathbf{I_k}$ de uma entidade também serão representadas como elementos independentes. A alteração do leiaute da interface de uma entidade somente será realizada para refletir mudanças significativas no estado interno da entidade, ou seja, mudanças no valor de seus atributos não computados. Assim, uma interface terá que ter conhecimento dos atributos dos quais ela depende. Além disso, será necessário que o usuário informe as condições sob as quais o leiaute da interface será modificado, ou seja, quais as mudanças de valores dos atributos provocam esta alteração. Por último, a interface deverá ter sua descrição armazenada (em um formato próprio para o sistema computacional no qual a aplicação estará operando). Esta forma de armazenamento, juntamente com a ação de **agentes de software construtores de interface** [MYERS '93], deverá possibilitar que a aplicação apresente as interfaces de uma entidade ao usuário na linguagem visual em que eles interagem e não na forma de um texto, normalmente ininteligível ao usuário final, facilitando assim sua elaboração e compreensão. A estrutura de uma interface na *ADKB* terá a seguinte forma:

```
EntityName-InterfaceName
        dependences     value           [attr_1, attr_2, …, attr_N]
                        constraints     {attr_i ∈ attributes}
        conditions      value           [cond_1, cond_2, …, cond_N]
                        constraints     {∀i,j cond_i ≠ cond_j}
        layouts         value           [[name_1 code_1], [name_2 code_2], …, [name_N code_N]]
                        constraints     {∀i,j code_i ≠ code_j}
```

Figura 23: Descrição da estrutura de uma interface na *ADKB*.

Na *ADKB*, as ações $\mathbf{B_i}$ das entidades serão igualmente representadas como elementos independentes. Elas deverão indicar quais **casos** (parâmetros nas linguagens de programação convencionais) o verbo (o nome da ação) que a descreve aceita, quais os mecanismos de ativação e os diálogos de interfaces ela emprega (que também deverão ser armazenados num formato próprio do sistema computacional, conforme citado acima) e qual conjunto de instruções será empregado

em sua realização. Estas instruções estarão representadas em uma linguagem de planejamento de IA para facilitar a geração de texto explicativo, conforme citado anteriormente. É possível representar os casos de um verbo como entidades independentes na *ADKB* e, com isto, acrescentar a eles modalidade (obrigatória ou opcional) e um valor *default*. Porém, a princípio, esta possibilidade não será explorada na linguagem-tipo para *EUP* aqui proposta devido à falta de estruturas lingüísticas semelhantes no subconjunto das sentenças imperativas da linguagem natural analisado, o que é um forte indicativo de que as pessoas não fazem uso deste tipo de mecanismo na expressão de planos no dia-a-dia.

De modo a apresentar uma estrutura de discurso coerente com o tipo de texto necessário a uma extensão, as ações também terão pré-condições que, normalmente, referem-se à existência de valores para seus parâmetros. A estas pré-condições poderão estar associadas ações, normalmente de acesso à interface, que serão ativadas para satisfazer as pré-condições, caso isto não ocorra quando da ativação da ação. Estas pré-condições estarão representadas por meio de condicionais *when* e poderão ser expressas juntamente com as instruções da ação. Elas poderão ser parcialmente geradas pelo sistema de extensão e, quando não o forem, irão requerer que o processo de realização de uma extensão, descrito no Capítulo 6, auxilie o usuário, lembrando-o da necessidade de sua especificação. Por último, as ações também terão um tratador de erros que permitirá emitir uma mensagem esclarecedora ao usuário do tipo de erro ocorrido, caso a ação tenha sido ativada pela interface, ou propagar o erro, caso a ação tenha sido ativada por outra ação. O tratador de erros também pode ser parcialmente inserido pelo sistema de extensão como as pré-condições e também poderá ser expresso juntamente com as instruções da ação, sendo representado por condicionais *when*. Assim, a estrutura de uma ação na *ADKB* terá a seguinte forma:

```
EntityName-ActionName
        preconditions   value           [cond1, cond2, …, condN]
                        constraints     {∀i,j condi ≠ condj}
        cases           value           [[prep1 refe1], [prep2 refe2] …, [prep4 refe4]]
                        constraints     {∀i,j [prepi refei] ≠ [prepj refej]}
        activators      value           [[name1 code1], [name2 code2] …, [nameN codeN]]
                        constraints     {∀i,j [namei codei] ≠ [namej codej]}
        dialogs         value           [[name1 code1], [name2 code2] …, [nameN codeN]]
                        constraints     {∀i,j [namei codei] ≠ [namej codej]}
        instructions    value           [inst1, inst2, …, instN]
        error-handling  value           (instruction)
```

Figura 24: Descrição da estrutura de uma ação na *ADKB*.

Também é possível definir relações diferentes das relações estruturais (*is-a* e *part-of*) na *ADKB*. Estas relações envolvem assertivas entre duas ou mais entidades e serão necessárias caso a linguagem-tipo para *EUP* tenha de suportar metáforas e metonímias. Do ponto de vista de uma entidade, uma relação será somente um novo *slot*, que armazenará os nomes das entidades a ela relacionadas, a ser inserido no *frame* que a representa na *ADKB*. Do ponto de vista da *ADKB*, ela será um elemento independente que deverá descrever seu nome e o nome de sua inversa e possibilitar o acesso a todos os elementos que estão relacionados por meio dela. Como as entidades, as relações poderão conter atributos e ações que poderão ser usados para realizar inferências sobre as assertivas que elas representam. Estas inferências podem ser empregadas para suportar mecanismos lingüísticos de referenciação mais elaborados como, por exemplo, o de metáforas e metonímias. Assim, a estrutura de uma relação não estrutural na *ADKB* terá a seguinte forma:

```
RelationName
       is-a           value          Relation
       inverse        value          (a String)
       cardinality    value          (1-1,1-n,n-m)
       source         value          [enti1, enti2, …, entiN]
       target         value          [enti1, enti2, …, entiM]
                      constraints    {∀i,j entii ≠ entij}
                                     {cardinality = 1-1 → M = 1}
                                     {cardinality = 1-n → M = n}
                                     {cardinality = n-m → M = m}
       attributes     value          (nil/[attr1, attr2, …, attrN])
                      constraints    {∀i,j attri ≠ attrj}
       actions        value          [acti1, acti2, …, actiN]
```

Figura 25: Descrição da estrutura de uma relação na *ADKB*

Uma vez descrita a estrutura proposta para a *ADKB*, apresentamos na Figura 26 uma gramática para a parte de metalinguagem da linguagem-tipo para *EUP* aqui proposta.

```
EXTENS → REFERE1[47] DECLM REFERE2 ENDM
DECLM  → 'is' | 'are'
EXTENS → REFERE1 DECLM REFERE2 (PRELM PROPM | PREPM) REFERE3 ENDM
PRELM  → 'that' | 'which' | 'whose'
PROPM  → 'has'
PREPM  → 'with'
EXTENS → REFERE1 DECLM PARTM REFERE2 ENDM
PARTM  → 'part of' | 'parts of'
EXTENS → REFERE DECLM STEPM ACTIONS[48] ENDM
```

[47] O não-terminal `REFERE` é descrito na gramática da Figura 19.

```
STEP_M   → 'the result of the instructions:' | 'follows these instructions:'
EXTENS → REFERE_1 PROP_M REFERE_2 END_M
EXTENS → REFERE_1 PROP_M REFERE_2 (SEP_M POSS_M VALU_M | PREL_M VALU_M DECL_M) REFERE_3 END_M
POSS_M   → 'with possible'
VALU_M   → 'value' | 'values'
EXTENS → REFERE_1 PROP_M REFERE_2 [SEP_M] PREL_M DECL_M REFERE_3 END_M
EXTENS → REFERE_1 PROP_M INTE_M REFERE_2 END_M
INTE_M   → 'an interface that depends on'
EXTENS → REFERE_1 PROP_M DEFV_M REFERE_2 END_M
DEFV_M   → 'default value'
EXTENS → HEADER DECL_M ACTI_M REFERE_1 [PREL_M ACTIVA [SEP_M STEP_M ACTIONS]] END_M
HEADER → ACTI_MI REFERE_1 [#PREP_1] REFERE_2 [#PREP_2 REFERE_3 [#PREP_3 REFERE_4 [#PREP_4 REFERE_5]]
ACTI_MI  → 'To'
ACTI_M   → 'an action of'
ACTIVA  → DECL_M ACTV_M WIDGET_1 [OPTI_M WIDGET_2 [OPTI_M WIDGET_3]]
ACTV_M   → 'activated by'
WIDGET  → 'menu' | 'shortcut' | 'button' | …
OPTI_M   → 'or by'
EXTENS → REFERE_1 ACTIVA [SEP_M STEP_M ACTIONS] END_M
EXTENS → REFERE_1 STEP_M ACTIONS END_M
```

Figura 26: Gramática que descreve os mecanismos de metalinguagem de uma *EUPL.*

A semântica para esta gramática será dada por um conjunto de funções auxiliares que criam novas entidades na *ADKB* e/ou acrescentam elementos às entidades já existentes. Assim, descreveremos a seguir um mapeamento semântico para as regras da gramática proposta, apresentando também alguns exemplos em uma *EUPL* instanciada da linguagem-tipo para *EUP* aqui proposta e o seu respectivo mapeamento. Deste modo, a regra:

```
EXTENS → REFERE_1 DECL_M REFERE_2 END_M
```

acrescentará uma nova entidade na *ADKB* mapeando **REFERE$_1$** (uma referência indefinida) para uma nova entidade $\mathbf{E_{j+1}}$ que **derivará** de uma entidade $\mathbf{E_k}$ pré-existente na *ADKB*, indicada por **REFERE$_2$** (uma referência indefinida). Assim, deverá ser gerada a expressão:

```
(new-concept REFERE_1 REFERE_2))
```

Se **REFERE$_2$** especificar um *container*[49], a nova entidade será considerada composta e a categoria de seus elementos deverá ser determinada, sendo adicionadas, para cada *NOME* em **REFERE$_1$**, as expressões:

---

[48] O não-terminal `ACTIONS` é descrito na gramática da Figura 20.

[49] Uma referência da forma *set of*, *list of* ou *sequence of.*

```
(set-compound NOME true)
(set-element-type NOME (get-container-type REFERE2))
```

como exemplos de uso desta regra temos:

***A** car **is a** vehicle.*

```
(new-concept car vehicle)
```

***An** address-book **is a** set **of** contacts.*

```
(new-concept address-book set)
(set-compound address-book true)
(set-element-type address-book (get-container-type "a set of contacts"))
```

A regra:

```
EXTENS → REFERE1 DECLM REFERE2 (PRELM PROPM | PREPM) REFERE3 ENDM
```

acrescentará uma nova entidade na *ADKB* e definirá novas propriedades (atributos e/ou partes) para esta entidade. Deste modo, ela mapeará **$REFERE_1$** (uma referência indefinida) para uma nova entidade $\mathbf{E_{j+1}}$ que derivará de uma entidade $\mathbf{E_k}$ pré-existente na *ADKB,* indicada por **$REFERE_2$** (uma referência indefinida). Em seguida, ela mapeará **$REFERE_3$** (uma lista de referências indefinidas) para uma **propriedade**[50] da nova entidade. Assim, será gerada a expressão:

```
(new-concept REFERE1 REFERE2)
```

e para cada *NOME* em $REFERE_3$ que se referir a uma entidade da *ADKB* deverá ser adicionada a expressão:

```
(add-part REFERE1 NOME)
```

e, caso o *NOME* não se referir a uma entidade, deverá ser adicionada a expressão:

```
(add-attribute REFERE1 NOME)
```

como exemplos de uso desta regra temos:

***A** contact **is a** concept **with a** nickname **and an** address-list.*[51]

```
(new-concept contact concept)
(add-attribute contact nickname)
(add-part contact address-list)
```

[50] A distinção entre atributo e parte será feita de forma automática, levando em conta o fato de que uma parte é sempre uma entidade já existente na *ADKB* e um atributo não.

[51] Onde *nickname* é um atributo e *address-list* um conceito da *ADKB*.

A regra:

```
EXTENS → REFERE1 PROPM REFERE2 ENDM
```

também acrescentará novas propriedades (atributos e/ou partes) a uma entidade pré-existente na *ADKB*. Deste modo, **REFERE$_1$** (uma referência indefinida ou uma referência anafórica interfrasal[52]) mapeará sobre uma entidade $\mathbf{E_k}$ pré-existente na *ADKB*, e **REFERE$_2$** (uma lista de referências indefinidas) mapeará sobre novos atributos $\mathbf{A_i}$ e/ou partes $\mathbf{P_j}$ desta entidade. Assim, para cada *NOME* em REFERE$_1$ se o *NOME* se referir a uma entidade da *ADKB*, deverá ser gerada a expressão:

```
(add-part REFERE1 NOME)
```

e, caso o *NOME* não se refira a uma entidade, deverá ser gerada a expressão:

```
(add-attribute REFERE1 NOME)
```

como exemplos de uso desta regra temos:

***A** note **has a** title**, a** date**, a** state **and a** text**.*[53]

```
(add-attribute note title)
(add-part note date)
(add-attribute note state)
(add-part note text)
```

A regra:

```
EXTENS → REFERE1 PROPM REFERE2 (SEPM POSSM VALUM | PRELM VALUM DECLM) REFERE3 ENDM
```

acrescentará novos atributos a uma entidade pré-existente na *ADKB* e definirá uma faixa de valores válidos para estes atributos. Deste modo, **REFERE$_1$** (uma referência indefinida ou uma referência anafórica interfrasal) mapeará sobre uma entidade $\mathbf{E_k}$ pré-existente na *ADKB* e **REFERE$_2$** (uma lista de referências indefinidas) mapeará sobre novos atributos $\mathbf{A_i}$ para esta entidade. A faixa de valores válidos para os atributos será definida pelos valores fornecidos por **REFERE$_3$** (uma lista de nomes próprios do tipo ' "*yellow*" ' ou ' "*green*", "*blue*" *and* "*yellow*" ', uma faixa de numerais na forma 1..45 ou um valor booleano "*true*" ou "*false*"), de modo que todos os atributos especificados na sentença terão a mesma faixa de valores. Assim, deverá ser gerada a expressão:

```
(add-attribute REFERE1 REFERE2)
```

e para cada *NOME* em REFERE$_1$ deverão ser adicionadas as expressões:

[52] Do tipo '*it*'. Esta anáfora deverá ser resolvida para uma entidade anteriormente referenciada no texto.

[53] *Title* e *state* são atributos. *Date* e *text* são partes.

```
(set-type NOME enumeration)
(set-value-range NOME REFERE₃)
```

como exemplos de uso desta regra temos:

***A** note **has a** state **with possible values** "hidden" **and** "visible".*

ou

***A** note **is a** concept. **It has a** state **whose values are** "hidden" **and** "visible".*

```
(add-attribute note state)
(set-type state enumeration)
(set-value-range state ("hidden" "visible"))
```

Como na regra anterior, a regra:

```
EXTENS → REFERE₁ PROP_M REFERE₂ [SEP_M] PREL_M DECL_M REFERE₃ END_M
```

acrescentará novos atributos a uma entidade pré-existente na *ADKB* e também definirá uma faixa de valores válidos para eles. Porém, nesta regra, a faixa de valores válidos será definida através da especificação da categoria à qual o atributo pertence, dada por $\mathbf{REFERE_3}$ (uma referência indefinida), de modo que todos os atributos especificados na sentença também terão o mesmo tipo. Assim, deverá ser gerada a expressão:

```
(add-attribute REFERE₁ REFERE₂)
```

e para cada *NOME* em $REFERE_1$ deverão ser adicionadas as expressões:

```
(set-type NOME category)
(set-value-range NOME REFERE₃)
```

como exemplos de uso desta regra temos:

***A** note **is a** concept. **It has a** title**, which is a** string.*

```
(add-attribute note title)
(set-type title category)
(set-value-range title string)
```

A regra:

```
EXTENS → REFERE₁ PROP_M DEFV_M REFERE₂ END_M
```

definirá o valor *default* de um atributo de uma entidade da *ADKB*. Deste forma, $\mathbf{REFERE_1}$ (uma referência definida ou uma referência anafórica) mapeará sobre um atributo $\mathbf{A_i}$ de uma entidade $\mathbf{E_k}$ pré-existente na *ADKB*. O nome desta entidade será determinado pelo **contexto** da extensão e $\mathbf{REFERE_2}$ (um nome próprio do tipo ' "*yellow*" ', um texto do tipo ' "*It's not possible*" ', um número 35

ou um valor booleano "*true*" ou "*false*") definirá o valor *default* do atributo. Este valor deverá estar dentro da faixa de valores válidos para o atributo. Assim, deverá ser gerada a expressão:

```
(set-default-value REFERE1 REFERE2)
```

como exemplos de uso desta regra temos:

***The** colour **has default value** "yellow"**.*
ou
***A** note **has a** colour**. It has default value** "yellow"**.*
`(set-default-value color "yellow")`

A regra:

```
EXTENS → REFERE DECLM STEPM ACTIONS ENDM
```

definirá as instruções que determinam o valor de um atributo computado de uma entidade da *ADKB*. Desta forma, **REFERE**$_1$ (uma referência definida ou uma referência anafórica) mapeará sobre um atributo $\mathbf{A_i}$ de uma entidade $\mathbf{E_k}$ pré-existente na *ADKB*. O nome desta entidade será determinado pelo **contexto** da extensão e **ACTIONS** fornecerá as instruções a serem executadas para o cálculo do valor do atributo, conforme descrito anteriormente nas regras da linguagem de controle. Deste modo, deverão ser geradas as expressões:

```
(set-computed REFERE true)
(add-instructions REFERE (interpret ACTIONS)))
```

como exemplos de uso desta regra temos:

*word-count **is the result of the instructions:** ...*
`(set-computed word-count true)`
`(add-instructions word-count (interpret "…"))`

A regra:

```
EXTENS → REFERE1 DECLM PARTM REFERE2 ENDM
```

acrescentará novas partes a uma entidade já existente na *ADKB*. Assim, **REFERE**$_1$ (uma lista de referências indefinidas ou uma referência anafórica interfrasal) mapeará sobre um conjunto de partes $\mathbf{P_i} \in \mathbf{E}$ que serão atribuídas a uma entidade $\mathbf{E_k}$ pré-existente na *ADKB,* indicada por **REFERE**$_2$ (uma referência indefinida). Logo, deverá ser gerada a expressão:

```
(add-part REFERE2 REFERE1)
```

como exemplos de uso desta regra temos:

*Date****, ****text* ***and*** *alarm* ***are parts of a*** *note****.***

```
(add-part note (date text alarm))
```

A regra:

```
EXTENS → REFERE1 PROPM INTEM REFERE2 ENDM
```

definirá quais atributos são responsáveis pela alteração da interface de uma entidade. Deste modo, **REFERE**$_1$ (uma referência indefinida ou uma referência anafórica interfrasal) mapeará sobre uma entidade $\mathbf{E_k}$ pré-existente na *ADKB*. A palavra reservada **interface** atuará como um *hyperlink* para a ativação dos agentes de software construtores de interface que auxiliarão o usuário na definição da interface obtendo dele a descrição completa da mesma. Assim, um agente de software será o responsável pela obtenção do nome da nova interface. Outro, agente será o responsável pela obtenção das condições de alteração da interface em função dos valores dos atributos descritos por **REFERE**$_2$ (uma lista de referência). E um terceiro, agente será responsável pela obtenção da definição do layout dos elementos da interface da entidade. Logo, deverá ser gerado o conjunto de expressões:

```
(new-interface call-interface-name-agent)
(set-dependence NOME REFERE2)
(set-conditions NOME call-condition-agent)
(set-layouts NOME call-layout-agent)
(add-interface REFERE1 NOME)
```

como exemplos de uso desta regra temos:

A *note* ***has an*** *interface* ***that depends on*** *colour****.***

ou

A *note* ***is a*** *concept****.*** ***It has an*** *interface* ***that depends on*** *colour****.***

```
(new-interface call-interface-name-agent)
(set-dependence NOME color)
(set-conditions NOME call-condition-agent)
(set-layout NOME call-layout-agent)
(add-interface note NOME)
```

A regra:

```
EXTENS → HEADER DECLM ACTIM REFERE1 [PRELM ACTIVA SEPM STEPM ACTIONS] ENDM
```

acrescentará uma nova ação a uma entidade pré-existente na *ADKB* e definirá a forma e os dados necessários para usá-la dentro de outra ação. Opcionalmente, ela possibilitará a definição da forma de ativação da ação através da interface e das instruções para sua execução. Deste modo, **HEADER**

fornecerá a forma e os dados necessários ao uso da ação. **REFERE₁** mapeará sobre uma entidade $\mathbf{E_k}$ pré-existente na *ADKB* à qual a ação pertencerá. **ACTIVA** fornecerá a formas de ativação da ação através da interface e **ACTIONS** fornecerá as instruções a serem executadas quando da sua ativação. Devido às opções existentes, esta regra pode gerar três diferentes conjuntos de expressões. O mais básico será:

```
(interpret HEADER)
(add-action REFERE1 NOME)
```

onde *NOME* corresponde ao nome do verbo que especifica a ação (nome este declarado no HEADER da ação). Com a adição do mecanismo de ativação pela interface, teremos:

```
(interpret HEADER)
(add-activators NOME (interpret ACTIVA))
(add-action REFERE1 NOME)
```

e com a adição da especificação das instruções teremos:

```
(interpret HEADER)
(add-activators NOME (interpret ACTIVA))
(add-instructions NOME (interpret ACTIONS))
(add-action REFERE1 NOME)
```

como exemplos de uso desta regra temos:

To** find <words> **is an action of the** agenda**.

```
(interpret "To find <words>")
(add-action agenda VERB)
```

To** find <words> **is an action of the** agenda **that is activated by a** menu**.

```
(interpret "To find <words>")
(add-activators VERB (menu menu-code))
(add-action agenda VERB)
```

***To** find <words> **is an action of the** agenda **that is activated by a** menu **and follows these instructions:** ...*

```
(interpret "To find <words>")
(add-activators VERB (menu menu-code))
(add-instructions VERB (interpret "…"))
(add-action agenda VERB)
```

A regra:

```
HEADER → ACTIMI REFERE1 [#PREP1] REFERE2 [#PREP2 REFERE3 [#PREP3 REFERE4 [#PREP4 REFERE5]]
```

definirá o nome da ação e o conjunto dos dados necessários a seu uso por outra ação. **REFERE₁** (uma referência indefinida) mapeará para uma nova ação $\mathbf{B_i}$ de uma entidade $\mathbf{E_k}$ (esta entidade será

especificada na regra que engloba a presente regra). Ela representa o nome do **verbo** que descreve a ação a ser realizada sobre os dados fornecidos. Os demais elementos **$REFERE_x$** (de 2 a 5) são referências indefinidas que mapearão sobre os **casos** do verbo (isto é, parâmetros) a serem empregados nas instruções da ação. Associadas às referências de 2 a 5 estão as preposições **$\#PREP_x$** (de 1 a 4) que marcam os casos válidos para o verbo dado por **$REFERE_1$**. Deste modo, deverá ser gerada a expressão:

```
(new-action REFERE1)
```

e, para cada caso especificado, deverá ser adicionada uma expressão:

```
(add-case REFERE1 ("PREPx" REFEREx))
```

como exemplos de uso desta regra temos:

***To** set **the** contact's <group> …*

```
(new-action set)
(add-case set ("" (part-of (var group) contact)))
```

***To** put down **all** <color> <objects> …*

```
(new-action put)
(add-case put ("down" (all (var color) (var object))))
```

***To** send **a** <message> **to a** <group> …*

```
(new-action send)
(add-case send ("" (var message)))
(add-case send ("to" (var group)))
```

***To** put **all** <color> <objects> from <**a** container> **at** <**a** place> …*

```
(new-action put)
(add-case put ("" (all (var color) (var object))))
(add-case put ("from" (var container)))
(add-case put ("at" (var place)))
```

To** change **the** color **of an** <object> **at** <place> **from** <color-1> **to** <color-2> **…

```
(new-action change)
(add-case change ("" (part-of (var object) color)))
(add-case change ("at" (var place)))
(add-case change ("from" (var color-1)))
(add-case change ("to" (var color-2)))
```

A regra:

```
EXTENS → REFERE1 ACTIVA [SEPM STEPM ACTIONS] ENDM
```

define as formas de ativação de uma ação através da interface e, opcionalmente, permite definir as instruções para execução quando da sua ativação. Deste modo, **$REFERE_1$** (uma referencia anafórica)

mapeará sobre uma entidade $\mathbf{E_k}$ pré-existente na *ADKB*. **ACTIVA** fornecerá a formas de ativação da ação através da interface e **ACTIONS** fornecerá as instruções a serem executadas quando da sua ativação. Logo, deverá ser gerada a expressão:

```
(add-activators REFERE1 (interpret ACTIVA))
```

e, no caso de serem especificadas as instruções, deverá ser adicionada a expressão:

```
(add-instructions REFERE1 (interpret ACTIONS))
```

como exemplos de uso desta regra temos:

***To** verb ... . It is* *activated* ***by a** menu **or by a** shortcut.*

`(add-activators verb (interpret (activated by menu and a shortcut)))`

***To** verb ... . It is* *activated* ***by a** menu **or by a** shortcut **and follows these instructions:** ....*

`(add-activators verb (interpret "activated by menu and a shortcut"))`

`(add-instructions verb (interpret "..."))`

A regra:

```
ACTIVA → DECLM ACTVM WIDGET1 [OPTIM WIDGET2 [OPTIM WIDGET3]]
```

especifica os mecanismos de ativação de uma ação através da interface. É possível definir a ativação por qualquer mecanismo suportado pelo sistema operativo no qual a aplicação executa. Assim, os WIDGET_x (de 1 a 3) representam os nomes dos mecanismos de ativação e atuarão como *hyperlinks* para a ativação de agentes de software construtores de interface que auxiliarão o usuário na criação e, quando necessário, na especificação do layout dos mecanismos de ativação selecionados. Assim, deverá ser gerada uma lista de expressões que contenha, para cada *widget* selecionado, uma expressão na forma:

```
(widget-name call-widget-name-creator-agent)
```

como exemplos de uso desta regra temos:

***To** verb ... . It is* *activated* ***by a** menu **or by a** shortcut...*

`((menu call-menu-creator-agent) (shortcut call-shortcut-creator-agent))`

O uso de diálogos de interação numa ação ocorrerá em duas situações:

1. Quando se deseja apresentar alguma informação ao usuário, caso em que deverá ser empregada a ação primitiva *show*; e

2. Quando se quer obter informação do usuário, caso em que deverá ser empregada a ação primitiva *ask*.

Na primeira situação, quando a informação a ser passada for uma mensagem simples será empregado um **diálogo padrão** que não requer sua especificação. Porém, quando a informação for mais complexa ou quando tratarmos da segunda situação, será necessário empregar **um diálogo projetado** para esta situação específica. Nesta situação, o usuário deverá especificar o diálogo usando um caso pré-definido para as duas ações primitivas anteriores:

```
("using" (name-of-dialog dialog))
```

No texto da extensão, o sintagma *name-of-dialog* dialog atuará como um *hyperlink* para o **agente de software construtor de diálogos** do sistema de extensão. Este agente auxiliará o usuário final na "criação do diálogo" e obterá dele o "leiaute" do diálogo. As informações que farão parte do diálogo irão compor os outros casos das duas funções primitivas. Assim, a especificação de diálogos nas instruções que compõem uma ação irá requerer que seja gerada a expressão:

```
(add-dialog action-name (name-of-dialog call-dialog-creator-agent)
```

Uma expressão igual a esta deverá ser adicionada, após a expressão de criação da ação, para cada diálogo presente nas instruções da ação.

A regra:

EXTENS → REFERE_1 STEP_M ACTIONS END_M

define as instruções para execução de uma ação. Deste modo, REFERE_1 (uma referência anafórica) mapeará sobre uma entidade $\mathbf{E_k}$ pré-existente na *ADKB* e ACTIONS fornecerá as instruções a serem executadas quando da ativação da ação. Deste modo, deverá ser gerada a expressão:

```
(add-instructions REFERE1 (interpret ACTIONS))
```

como exemplos de uso desta regra temos:

To *verb ….* ***It follows these instructions****: ….*
(add-instructions verb (interpret "…"))

Para melhor exemplificar estes mecanismos é apresentada na Figura 27 uma extensão, codificada em uma *EUPL* instanciada de linguagem-tipo para *EUP* aqui proposta, para o domínio de um *E-mail client* simples cujo modelo de domínio é apresentado Figura 28. Um ponto muito importante a ser observado neste modelo é que ele expressa somente os elementos visíveis ao

usuário final através da interface do software, ou seja, ele é o modelo de usabilidade do software percebido pelo usuário. Além disso, é relevante observar que o modelo também expressa as funções primitivas do sistema para as quais a explicação é composta de um texto pré-definido pelo designer.

```
% Extension code for a behaviour to send a <message> to a <group>

To send a <message> to a <group> is an action of the agenda. It is
activated by the menu and follows these instructions:

If the <group> belongs to the address-book, send the <message> to all
contacts of the <group>.

If not, show the message "The specified group does not exist. Please,
check your data and try again." using the error-handling dialog.

When the <message> or the <group> are not indicated, ask for them using
the get-data dialog.

When there is any error, show the message "Some error has occurred
during the sending of the messages. Please, check your data and try it
again" using the error-handling dialog.
```

Figura 27: Exemplo de texto de uma extensão na *EUPL*.

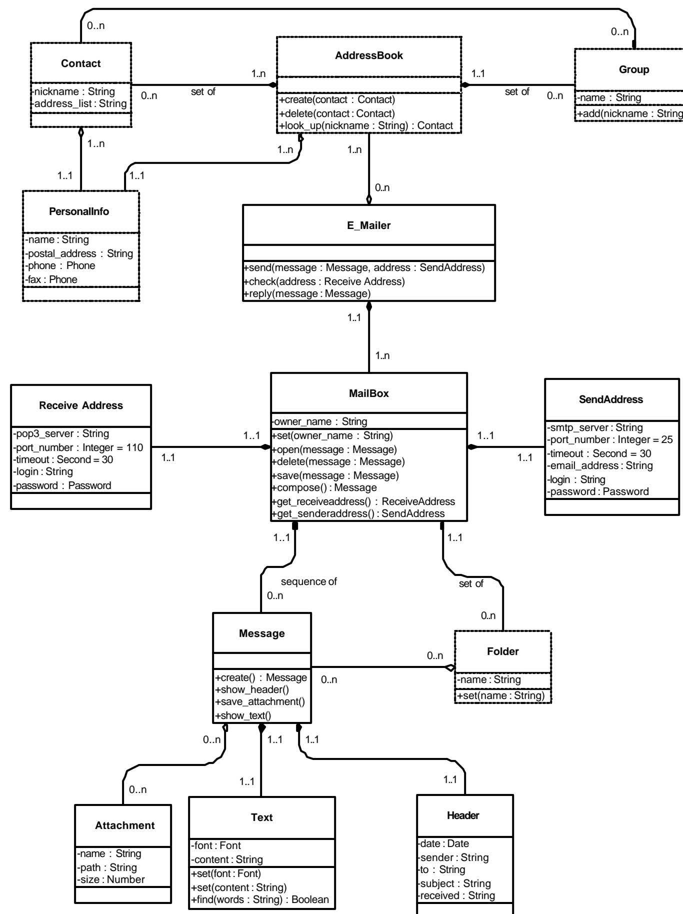

Figura 28: Modelo do domínio para um cliente de *e-mail* simples.

Na extensão apresentada na Figura 27, a sentença que começa com *To send a ...* descreve a parte da metalinguagem que acrescenta uma nova ação em uma entidade do sistema; as demais partes descrevem o texto da *EUPL* que descreve as atividades a serem realizadas nesta ação, fazendo o uso das sub-linguagens de referência e de controle descritas. Exemplos mais completos são apresentados no Anexo II deste trabalho.

É interessante observar alguns pontos no texto da Figura 27. Primeiro, toda sentença que apresente um "`%`" indica um comentário que se estenderá até o final da linha. As instâncias de referências marcadas por "<" e ">" como, por exemplo, "`<message>`" indicam **variáveis** do tipo do conceito referenciado, no presente caso "*message*". Outro ponto interessante é que, apesar de o texto não empregar mecanismo de repetição explícito, ele realiza uma iteração sobre os elementos do tipo "*contact*" através da referência a todos seus elementos ("`all contacts`"), como pode ser visto em "`send the <message> to all contacts of the <group>`". Como dissemos anteriormente, este tipo de iteração implícita é o meio mais comum de realização de iterações entre os usuários finais, como também nos aponta Pane *et al.* [PANE '01]. A Figura 29 apresenta o mapeamento semântico para o texto da Figura 27.

```
%      Translation code for an action to send a <message> to a <group>
%
(new-action send)
(add-case send ("" (var message))
(add-case send ("to" (var address-book))
(add-action agenda send)
(menu call-menu-creator-agent)
(add-activations send menu)
(add-instructions send (
 ((belongs-to? address-book (var group))
  (on-true (iterate (apply send (var message) (part-of (var group) contact)) last))
  (on-false (apply show (message "The specified group does not exist. Please, check
        your data and try again.") (using error-handling)))
))
(add-preconditions send (
 ((and (not (indicated? (var message))) (not (indicated? (var group))))
  (on-true (get-data ((var message) (var group)) (using get-data))))
))
(add-error-handling send (
 (apply show (message "Some error has occurred during the sending of the messages.
        Please, check your data and try it again") (using error-handling))
))
```

Figura 29: Mapeamento semântico do corpo da ação especificada pela extensão da Figura 27.

2.2. Sobre a expressividade das *EUPLs* derivadas da linguagem-tipo para *EUP* proposta

Para avaliarmos a expressividade de uma *EUPL* gerada a partir da linguagem-tipo para *EUP* proposta no Modelo Semiótico é importante observarmos seus compromissos ontológicos (o que elas permitem expressar) e epistemológicos (o que se pode conhecer sobre o que elas permitem expressar).

O primeiro compromisso ontológico destas *EUPLs* está relacionado à sua especificidade no domínio da aplicação extensível na qual ela está embutida. Este compromisso impõe uma redução em seu universo de discurso limitando, deste modo, seu poder expressivo. Tal limitação visa impedir que um usuário final possa criar extensões à aplicação que sejam totalmente novas e que possam vir a subverter a mensagem original do designer do software. Ao mesmo tempo, ela auxilia a esclarecer a distinção feita, na abordagem da Engenharia Semiótica, entre o que é uma extensão (a adição de novos elementos a um domínio de discurso já existente) e o que é programação (a criação de um novo domínio de discurso).

É relevante notar que o maior diferencial destas *EUPLs* com relação as LPs atuais está em sua estrita relação com a *UIL* da aplicação extensível na qual ela está embutida. Conforme de Souza *et al.* [DE SOUZA' 01], a linguagem de interação com o usurário de uma aplicação (sua *UIL*) é única, sendo que esta unicidade advém do dialeto de interação criado pelo designer do software durante sua construção. Este dialeto será, por sua vez, definido pelo conjunto de elementos de interação escolhidos (dentre os disponibilizados pelo sistema computacional em que a aplicação se insere) e pela forma única de combiná-los (na aplicação em questão) definidos pelo designer do software.

Deste modo, o segundo compromisso ontológico destas *EUPLs* está relacionado com sua restrição sobre o conjunto de elementos interativos disponíveis aos usuários finais para a criação de diálogos interativos na realização de novas extensões. Este compromisso também opera no sentido de restringir o universo de discurso destas *EUPLs* e, portanto, limitar seu poder expressivo. Tal restrição resulta do fato de, no Modelo Semiótico, a relação entre a *EUPL* e a *UIL* de uma aplicação extensível ser delimitada pelo princípio do Contínuo Semiótico. Como já vimos, este princípio requer inicialmente que a *UILx* (a parte extensível da *UIL*) seja uma Abstração Interpretativa da *EUPL*. Sendo assim, a *EUPL* deve prover descrições semânticas para a *UILx* da aplicação na qual ela está embutida, mas um usuário não precisa conhecer a *EUPL* para entender os signos da *UIL*. Assim sendo, este requisito impõe a necessidade de que uma *EUPL* represente

discursos sobre os elementos interativos presentes no dialeto da *UILx* da aplicação extensível na qual ela será embutida, requerendo que os elementos interativos empregados na *UIL* da aplicação façam parte do domínio expressivo de sua *EUPL*.

Entretanto, o aspecto mais importante do princípio do Contínuo Semiótico é o compromisso epistemológico que ele impõe sobre estas *EUPLs*, subtraindo dos textos possíveis de serem gerados pela suas gramáticas aqueles que não são válidos por não produzirem efeitos situados na linguagem única de interação da aplicação. Este compromisso essencial é resultado da exigência do princípio do Contínuo Semiótico de que todo "texto" gerado pela *EUPL* de uma aplicação extensível seja pragmaticamente válido. Conforme discutido anteriormente, um "texto" será pragmaticamente válido em uma *EUPL* se ele apresentar uma representação válida, seja ela realizada ou potencial, situada na *UIL* da aplicação na qual ela está embutida.

Portanto, o compromisso epistemológico atua como fator primordial na definição do poder expressivo destas linguagens, eliminado a possibilidade de os usuários finais gerarem extensões que não atuem sobre a *UIL* da aplicação. Além disso, ele também elimina a possibilidade de os usuários construírem extensões que apresentem elementos interativos diferentes daqueles escolhidos pelo designer para comporem a interface da aplicação em questão, evitando assim que se corrompa a mensagem original do designer.

Logo, as *EUPLs* geradas a partir da linguagem-tipo para *EUP* proposta no Modelo Semiótico apresentam um poder expressivo menor que o das LP atuais, pois restringem a ação de "programação" por parte do usuário final às alterações sobre os elementos do domínio da aplicação extensível que apresentam alguma representação na *UIL* desta aplicação. É extremamente importante salientar que é exatamente esta limitação que concede ao Modelo Semiótico sua melhor adequação à descrição da tarefa de *EUP*, em relação aos demais paradigmas citados no Capítulo 2, na medida em que ele permite ao usuário criar suas extensões sem possibilitar que estas venham a corromper a mensagem original do designer do software.

A garantia de realização dos compromissos acima descritos requer a manutenção do Contínuo Semiótico entre as linguagens que atuam em uma aplicação extensível, tarefa esta somente realizável por construção. Ou seja, é necessário que ao se criar a aplicação os designers e engenheiros de software definam uma base de conhecimento do design da aplicação (a *ADKB*) que contenha o núcleo inicial de apoio à manutenção do Contínuo Semiótico entre as linguagens da aplicação. Tal base, conforme discutido neste capítulo, deverá conter os modelos do domínio da

aplicação e do sistema computacional em que a aplicação está inserida (em especial a parte que descreve seus elementos de interação) e também o conjunto de regras de design empregadas na definição da linguagem única de interação da aplicação.

Partindo da *ADKB* inicial, o processo de realização de extensões definido pelo Modelo Semiótico, que será discutido no Capítulo 6, irá fazer uso de consultas a *ADKB* para garantir a pertinência dos elementos referenciados no texto da extensão ao universo de discurso definido na *ADKB*, forçando assim o respeito aos compromissos ontológicos acima discutidos. A garantia de respeito ao compromisso epistemológico imposto pelo princípio do Contínuo Semiótico será obtida através de duas fontes: a própria estrutura do processo de realização de extensões, que garante a manutenção do Ciclo Mínimo de Interação da *UIL*, e o emprego de agentes inteligentes de software, que fazem uso das regras de design armazenadas na *ADKB*, para orientar os usuários finais na construção de novos elementos de interação. A função dos agentes de software é suprimir a criação de elementos iterativos estranhos ao dialeto da *UIL* da aplicação, impossibilitando a descrição de uma entidade totalmente nova na aplicação e assegurando o compromisso epistemológico definido.

Deste modo, encerramos a descrição da linguagem-tipo para *EUP* aqui proposta e dos mecanismos necessários a seu funcionamento e à manutenção dos princípios da Abstração Interpretativa e do Contínuo Semiótico. No capítulo a seguir apresentaremos o processo de realização de uma extensão pelos usuários finais.

Capítulo 6

A ORGANIZAÇÃO DA TAREFA DE *EUP*

No Capítulo 3 deste trabalho, defendemos a necessidade da inclusão de dois novos componentes nas arquitetura de software (a saber, um ambiente de extensão e uma linguagem de extensão) como requisito mínimo para que um software possa permitir a um usuário a realização de extensões. Defendemos também que, além destes componentes, são necessários outros requisitos para apoiar o usuário na aquisição de conhecimento sobre o uso do software e de sua linguagem de extensão e para regular a forma das extensões que podem ser realizadas a fim de preservar o conhecimento por ele adquirido. No Capítulo 5, descrevemos a estrutura de uma arquitetura de software que dá apoio à vigência dos dois princípios da Engenharia Semiótica observáveis nas extensões criadas, ajudando na manutenção do conhecimento adquirido pelo usuário. Descrevemos também uma linguagem-tipo para *EUP* que satisfaz aos princípios e requisitos levantados no Capítulo 3 e que, além disso, apresenta um rico conjunto de mecanismos comunicativos. Todos estes elementos são imprescindíveis à criação de um modelo baseado em Semiótica para a tarefa de *EUP*. Resta-nos agora definir o modo pelo qual estes elementos se inter-relacionam dentro de um processo de realização de extensões pelos usuários finais.

É importante lembrar que a linguagem-tipo para *EUP* descrita no Capítulo 5 e o processo de criação de extensões, a ser descrito neste capítulo, estão intimamente ligados, pois é necessário que alguns requisitos do processo sejam projetados como restrição sintática para a formação de sentenças e textos de uma *EUPL* que instancie esta linguagem-tipo.

1. O processo global de realização de uma extensão

A organização do processo que compõe o Modelo Semiótico para a tarefa de *EUP* será descrita por meio de um conjunto de diagramas de atividades[54] que representam algoritmos de alto nível que operam sobre a *ADKB* e tratam da definição de novas ações, entidades, atributos, relações e elementos representacionais a serem inseridos no modelo da aplicação. As relações procuram capturar os inter-relacionamentos entre as entidades do domínio da aplicação (elas serão particularmente úteis na resolução de referências e analogias empregadas na *UIL* e na *EUPL*).

No processo aqui proposto, o usuário, ao iniciar a tarefa de extensão, passará a fazer uso de um novo contexto de trabalho — um ambiente de extensão. Este ambiente emprega uma linguagem de interação específica, que integra a *UIL* da aplicação, por meio da qual o usuário poderá descrever as ações metalingüísticas que definirão sua extensão. Para auxiliá-lo, e para satisfazer os requisitos e princípios citados anteriormente, este ambiente deverá fornecer mecanismos de suporte específicos à realização da tarefa de extensão. Dentre estes mecanismos, o principal será um agente de software no formato de uma **Agenda** que apresentará ao usuário a seqüência de etapas necessárias à realização de sua extensão. Esta agenda será atualizada dinamicamente durante o processo, de modo a refletir as atividades de cada etapa. O ambiente de extensão também deverá apresentar as opções disponíveis à realização das atividades em cada etapa, ressaltando as funções da linguagem mais importantes para a compreensão da atividade em questão.

É importante relembrar que, pela interpretação dada na Seção 1.2 do Capítulo 3, no Modelo Semiótico **toda extensão a um software deve alterar sua *UIL***. Assim, a realização de uma extensão sempre requer a criação, modificação ou revogação[55] de um ou mais elementos de interação da *UIL*. Deste modo, o processo global de realização de uma extensão, segundo este

[54] Estes diagramas são uma variante dos diagramas de atividades da linguagem gráfica de especificação *UML*.

[55] A palavra **revogação** empregada neste capítulo é usada com licença de expressão visto ela implicar um encolhimento e não uma extensão. Além disso, é importante ressaltar que a revogação aqui discutida é relativa às extensões criadas pelos usuários, visto que, pelos princípios da Engenharia Semiótica, a revogação do design original da aplicação não é aceitável.

modelo, pode ser representado pelo diagrama de atividades[56] da Figura 30, sendo dividido em seis etapas principais:

1. A **especificação do tipo** da extensão — por meio da seleção entre as tarefas de criação, modificação ou revogação de funcionalidades do software;
2. A **realização** da extensão selecionada — por meio de um conjunto específico de tarefas;
3. A **validação sintática** da extensão — por meio da verificação da consistência do texto da extensão;
4. A **validação semântica** da extensão — por meio do teste da extensão com participação direta do usuário;
5. A **documentação** da extensão — por meio de um diálogo direto com o usuário; e
6. A **atualização da *ADKB*** — por meio do armazenamento ou remoção automática de elementos da *ADKB*.

Todas estas etapas são essenciais à realização de uma extensão que respeite os requisitos e princípios definidos nos Capítulos 3 e 5.

A tarefa de extensão que o usuário quer realizar sobre a aplicação, especificada na etapa 1[57], poderá ser a de: criar uma nova extensão, modificar uma extensão existente ou revogar uma extensão existente. Esta tarefa será realizada na etapa 2, por meio de três atividades mutuamente excludentes. A etapa 3 será empregada somente para os casos de criação e modificação. Ela verificará se os elementos do texto da descrição da extensão respeitam as regras sintáticas da *EUPL* e os requisitos levantados nas seções anteriores. A etapa 4 verificará se a descrição da extensão realizada é válida, requerendo para isto a intervenção direta do usuário. Esta intervenção é necessária uma vez que, conforme nos mostra Fetzer [FETZER '88], os programas, quando vistos como a codificação de algoritmos que podem ser compilados e executados em uma máquina, somente podem ser submetidos a uma verificação relativa e apenas o usuário sabe qual a intenção real para sua extensão.

---

[56] As figuras tracejadas encontradas nestes diagramas indicam atividades nas quais o usuário não pode interferir diretamente e as coloridas indicam atividades que apresentam sub-atividades.

[57] Os estados que pertencem a uma etapa estão marcados com o número da etapa no início de seu nome.

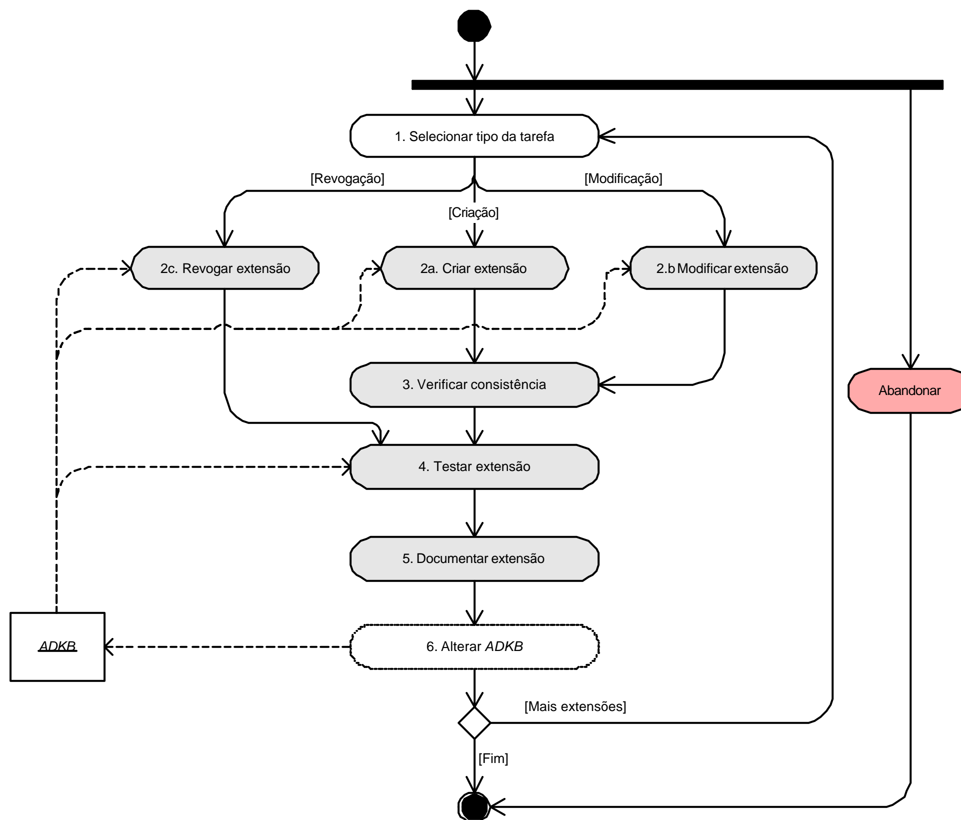

Figura 30: Processo global de realização de extensões em um software extensível segundo o Modelo Semiótico proposto.

Por meio das etapas 3 e 4 o processo procura garantir, até onde for possível[58], que toda extensão a ser inserida na *ADKB* esteja correta e seja válida e, portanto, poderá ser empregada na aplicação sem riscos para o trabalho do usuário. A etapa 5 realizará a documentação da extensão e será a responsável direta pelo grão mínimo de explicação disponibilizado pela *UEL* do software para a nova funcionalidade nele inserida. Esta explicação é livremente expressa pelo usuário, não sendo possível garantir que ela esteja condizente com a função realizada pela extensão. A etapa 6 atualizará a *ADKB* de forma automática e, juntamente com as etapas de validação e documentação, garantirá a sua evolução consistente durante o processo de realização da extensão. O tipo de atualização realizada dependerá do tipo de extensão que o usuário realizou. Deste modo, a *ADKB* poderá ser empregada futuramente pela *UEL* na geração de explicações ao usuário sobre a extensão que acabou de ser realizada.

É importante notar que o processo prevê a possibilidade de o usuário querer **abandonar** a tarefa de realização de uma extensão a qualquer momento de sua realização. Todavia, será

[58] Nem sempre é possível garantir que todas as alternativas de uma extensão foram testadas em todos os possíveis contextos.

necessário que esta informação seja marcada na *UIL*, por meio de mecanismos que ressaltem sua função fática, de modo que o usuário consiga sempre perceber esta possibilidade.

As etapas de 2 a 4 são etapas mais complexas e serão mais bem detalhadas a seguir.

2. A etapa de realização da extensão

Na etapa de realização de uma extensão, as atividades de criação, modificação e revogação poderão operar sobre ações, entidades, atributos, relações e elementos representacionais presentes no domínio da aplicação. Neste ponto, é importante ressaltar que, no Modelo Semiótico para tarefa de *EUP*, **todas as funcionalidades de um software estão ligadas a uma entidade na *ADKB***. Deste modo, ao realizar uma extensão sobre uma entidade já existente na *ADKB* será sempre minimamente necessário criar, modificar ou revogar uma ou mais ações de uma ou mais entidades que compõem o modelo de domínio do software. Conseqüentemente, as ações serão os elementos mais freqüentemente estendidos pelos usuários finais. Entretanto, uma extensão que implique a criação de novas entidades e/ou relações na *ADKB* requer que o usuário tenha maior conhecimento do modelo de usabilidade do software para ser capaz de inferir a necessidade de estender não somente uma ação mas, também, de alterar a estrutura do próprio modelo da aplicação para que ele possa atingir seu objetivo.

2.1. O processo de criação de uma nova ação

O processo de criação de uma nova ação para uma entidade descreve a estrutura de discurso necessária à completa descrição desta ação e está apresentado no diagrama de atividades da Figura 31, sendo composto de sete etapas:

1. A declaração do seu **nome**;
2. A definição de sua **sintaxe de uso** (especificando as entidades sobre as quais ela atua);
3. A sua **associação** com uma entidade da *ADKB*;
4. A declaração de sua **interface de ativação pela *UIL*** (por meio de uma tarefa própria para a edição de elementos da *UIL* relacionados aos canais de entrada de dados);
5. A definição de um **diálogo de aquisição de dados** para o caso de sua ativação pela *UIL* (por meio de uma tarefa própria para a edição de diálogos);

6. A definição de seu **conjunto de comandos** (por meio de uma tarefa própria para especificação de sua ordenação); e
7. A definição de uma **mensagem de erro** (para o caso em que ela falhe durante sua execução).

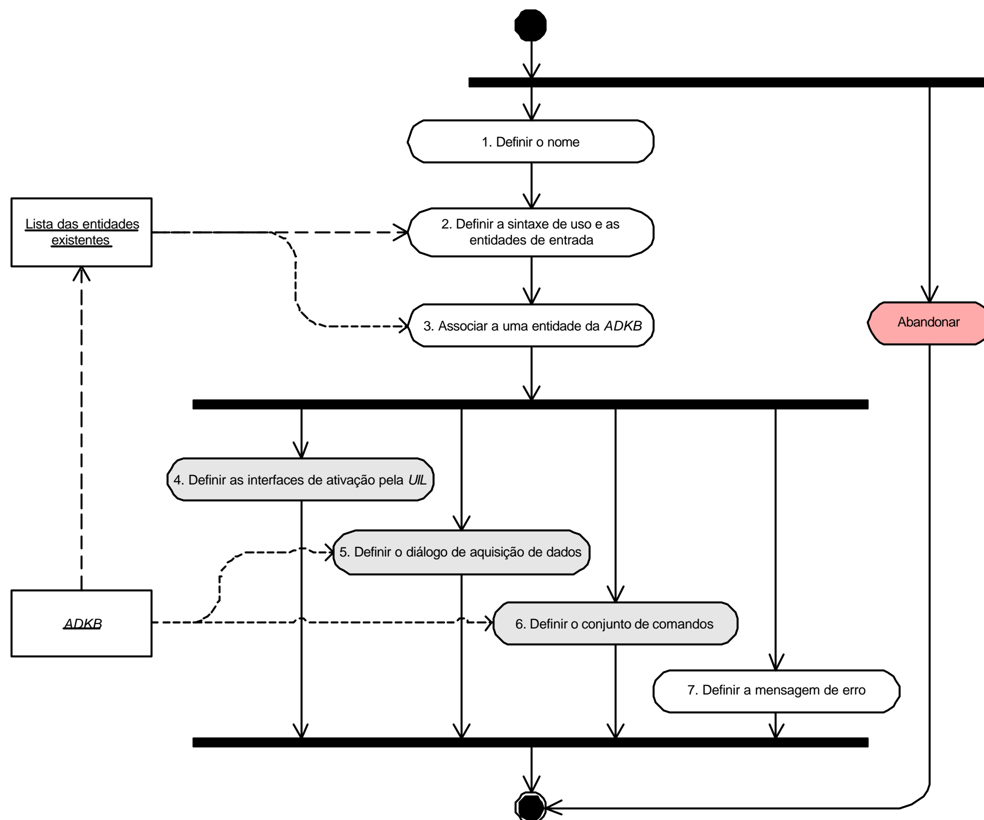

Figura 31: Descrição do processo de criação de novas ações em um software extensível.

Todas as etapas deste processo são essenciais à definição de uma ação que siga os requisitos e princípios definidos no Modelo Semiótico.

É relevante ressaltar que a estrutura deste processo é resultante de testes com uma versão preliminar de uma instância da linguagem-tipo para *EUP* proposta no Capítulo 5 deste trabalho (apresentados no Anexo I). Estes testes sugeriram que tanto os usuários leigos quanto os programadores concentram-se primordialmente na descrição do processamento que deve ocorrer em uma ação, sem atentar para a descrição dos comandos de **aquisição de dados** e de **apresentação de resultados**, assim como de **tratamento de erros**. É muito importante observar que este comportamento está de pleno acordo com o foco do discurso em questão, isto é, a descrição das atividades que irão transformar o estado atual do software no estado de interesse do

usuário. Deste modo, em respeito ao princípio da cooperação* postulado por Grice [GRICE '75], os usuários não mencionarão nada a mais do que for necessário à realização das atividades em questão. Em vista disso, fica claro que o ambiente é que deverá **topicalizar*** e **focar** os elementos do processo que, embora importantes computacionalmente, não são assim considerados pelos usuários ou programadores, que os relegam ao plano das pressuposições* e implicaturas*. Logo, fica clara a necessidade de empregar a função conativa da *UIL* do ambiente de extensão a fim de garantir que as etapas 5 e 7 sejam realizadas. A etapa 6 também irá requerer alterações em sua estrutura, conforme será visto mais adiante.

Como vimos no Capítulo 5, o Modelo Semiótico possibilita dois níveis de uso de uma ação. Assim sendo, uma ação poderá ser empregada como **uma funcionalidade associada à *UIL* do software** ou como **uma função usada internamente à outra ação** na *EUPL*. Estas duas formas procuram refletir os diferentes níveis de conhecimento da tarefa de *EUP* que um usuário detém. Assim, a primeira requer simplesmente que ele saiba que toda ação está associada a um elemento da *UIL*. No entanto, a segunda requer que ele infira a possibilidade de reutilização de uma ação dentro de outra (o que requer maior conhecimento computacional do usuário). Para viabilizar estes mecanismos, é necessário que o usuário possa definir não somente a forma de ativação da ação a partir da *UIL* mas, também, sua interface com outra ação, ou seja, sua sintaxe de uso dentro de um texto da *EUPL*.

Dado o nível de conhecimento necessário às duas formas de uso de uma ação, é plausível imaginar que o usuário poderia declarar primeiro a forma de ativação da ação pela *UIL* e, depois, definir sua sintaxe de uso em um texto da *EUPL*. Todavia, no Modelo Semiótico, o usuário definirá primeiro a sintaxe de uso da ação nos textos da *EUPL*, depois o nome da entidade a qual esta ação estará associada na *ADKB* e, por último, sua forma de ativação pela *UIL*. A razão para esta seqüência, marcada pela ordenação das etapas 2, 3 e 4, está no fato de ela gerar uma sentença escrita mais bem formada do ponto de vista lingüístico e, portanto, de maior facilidade de interpretação e aprendizado pelo usuário final.

O nome da nova ação que é definido na etapa 1 não tem necessariamente de ser único, pois o que define uma ação dentro de uma entidade é a sentença formada pela associação do seu nome com sua sintaxe de uso nos textos da *EUPL* (o que é equivalente à assinatura de um procedimento nas linguagens de programação). Deste modo, nós admitimos uma forma simples de **polissemia*** na *EUPL* (que é equivalente a algumas formas de **polimorfismo** em linguagens de programação).

Na etapa 2, a comunicação do usuário com o software estará centrada na função referencial da *UIL*, quando ele nomeia as entidades que serão manipuladas pela ação; e na função metalingüística da *EUPL*, quando ele define a forma de uso da ação nos textos da *EUPL* (isto é, a ordem das entidades e as preposições que as unirão na formação de uma sentença escrita na *EUPL*). Por meio da listagem das entidades da *ADKB* que podem ser utilizadas na definição das entidades manipuladas, o ambiente de extensão deverá auxiliar e restringir a atuação do usuário de modo a garantir a consistência sintática da extensão. Esta etapa está indiretamente relacionada ao mecanismo de manutenção do princípio do Contínuo Semiótico entre a *EUPL*, a *UILx* e a *UEL*, uma vez que as entidades necessárias ao uso da extensão pela *EUPL* e pela *UIL* são as mesmas, como veremos na etapa 5.

É interessante notar que a possibilidade de definir as preposições que ligam o verbo que dá o nome a ação e as entidades por ele manipuladas produz alterações sintáticas na própria *EUPL*. Isso faz com que a linguagem-tipo para *EUP* proposta neste trabalho não apresente uma sintaxe única para a "aplicação de um procedimento", como é comum nas linguagens convencionais. Deste modo, as *EUPLs* instanciadas desta linguagem-tipo serão linguagens com características dinâmica e reflexiva sem, no entanto, gerar problemas para seu *parser*, uma vez que o conjunto de preposições existentes na LN é fechado, restringindo, portanto, as opções de escolha do usuário. Esta é uma abordagem que permite maior legibilidade para a sintaxe de uso de uma ação facilitando, deste modo, a interpretação do texto da extensão produzida pelo usuário final.

A associação da ação que está sendo criada a uma entidade da *ADKB*, realizada na etapa 3, é responsável por manter a estrutura da *ADKB* e por garantir o respeito ao requisito de realização de extensões monotônicas, uma vez que ela não permite a criação de elementos isolados dentro do modelo da aplicação. Nesta etapa, a comunicação do usuário com o software estará centrada na função referencial, sendo que o ambiente de extensão deverá auxiliá-lo na navegação pelo modelo da aplicação, por meio da apresentação das entidades da *ADKB* possíveis de serem empregadas na associação.

É interessante notar que as etapas de 4 a 7 não apresentam uma ordem explícita de realização. Na etapa 4, o diálogo do usuário com o software estará situado na função poética da *UIL*, procurando definir um ou mais elementos de interação que poderão ativar a nova ação pela *UIL*. Logo, esta etapa é responsável por garantir que toda ação descrita na *EUPL* tenha obrigatoriamente um reflexo na *UIL*, fazendo parte do mecanismo de manutenção dos princípios

de Abstração Interpretativa e do Contínuo Semiótico. Como o usuário poderá associar um signo para cada um dos diferentes canais de entrada de dados aceito pelo software como, por exemplo, um atalho de teclado, uma entrada no menu, um comando de voz etc., será necessário definir um agente de software de construção de interface específico para restringir sua ação na realização desta tarefa. Tal restrição é necessária para garantir a manutenção da função poética da *UIL* do software (ou seja, sua aparência). Para realizar esta restrição, será necessário utilizar as regras de design (armazenadas na *ADKB*) empregadas pelo designer no desenvolvimento do software, a fim de filtrar as possibilidades de alteração da *UIL* de que o usuário dispõe. Ao mesmo tempo, este agente estará auxiliando o usuário na realização desta tarefa, visto ele não possuir conhecimento computacional suficiente para isto. Por exemplo, no caso do uso de um menu o usuário deverá ser auxiliado por meio de um **agente de software para edição de menus** que pré-selecionará a posição no menu em função das características da entidade a qual a nova ação está associada.

A etapa 5 realiza a definição do diálogo de aquisição de dados para as entidades que a ação manipula quando ativada pela *UIL*. Logo, nesta etapa, assim como na anterior, a comunicação do usuário com o software também estará centrada na função poética da *UIL*. Como dissemos anteriormente, as entidades manipuladas nesta etapa são as mesmas que as manipuladas na etapa 2. Por conseguinte, o texto da *EUPL* referente a esta etapa pode ser totalmente inferido a partir da descrição da sintaxe de uso da ação em textos da *EUPL*, evitando assim a necessidade de descrição pelo usuário. Então, restará ao usuário somente a definição da expressão deste diálogo na *UIL*, uma vez que seu conteúdo já está pré-fixado. A existência deste dialogo é fundamental para que a ativação de uma ação pela *UIL* esteja completa, já que sem ele os dados necessários à sua execução estarão indisponíveis. Deste modo, esta etapa é essencial para garantir o Ciclo Mínimo de Interação nesta nova ação. Assim, a agenda será acrescida de um novo processo constituído de quatro atividades:

- A **seleção de um objeto** (a intenção comunicativa do usuário) a ser representado no diálogo;
- A **seleção de uma representação** — um signo — (um *widget* de interface) para este objeto;
- A **inserção da representação na estrutura sintática** do diálogo; e
- A **documentação do uso** do objeto no diálogo (o registro da intenção de comunicação do usuário).

Esta seqüência de atividades é essencial à correta construção de um diálogo de interface pelo usuário. O ponto crucial neste processo está na seleção dos *widgets* de interface (os signos que representarão as entidades) a serem empregados na construção do diálogo. Para garantir a manutenção da função poética da *UIL*, o ambiente de extensão deverá restringir o conjunto dos *widgets* possíveis de serem empregados pelo usuário na construção deste diálogo utilizando para isto um **agente de software construtores de diálogos**. Este agente também contemplará o requisito de auxiliar o usuário a detectar os textos da *EUPL* que não são válidos na *UIL*. Tal objetivo será alcançado pela exclusão deste tipo de texto, inibindo sua criação por meio do uso de elementos de interação predefinidos pelo designer do software, que serão determinados pela interseção das mesmas regras de design empregadas na etapa 4 do processo de criação de uma nova ação com as entidades manipuladas. Além disto, ao vincular os elementos do diálogo (que fazem parte da *UIL*) às entidades manipuladas pela ação (que fazem parte da *EUPL*) garantimos a manutenção do Contínuo Semiótico entre a *EUPL* e a *UILx*.

Retomando as etapas do processo de criação da nova ação, a etapa 6 definirá o conjunto de comandos a serem desempenhados pela ação. Para isto, a agenda será acrescida de um novo processo constituído de três atividades:

- A **escolha do comando** a ser realizado;
- A **descrição do comando** (completando os elementos que o descrevem); e
- A definição de um **diálogo de saída de dados** para o comando (caso seja necessário passar resultados ao usuário).

As atividades 1 e 2 são essenciais a esse processo, constituindo a parte básica da descrição de uma ação. O conjunto de comandos disponíveis para serem utilizados nestas atividades, assim como sua estrutura, foram descritos no Capítulo 5. Estas atividades exigem conhecimento tanto da gramática quanto da semântica da *EUPL*. A fim de facilitar a aquisição deste conhecimento, o ambiente de extensão deverá disponibilizar ao usuário um conjunto de *templates* de comandos que possam ser selecionadas e adicionadas ao editor de ações do ambiente. Deste modo, na atividade 1, ao invés de precisar relembrar a sintaxe do comando, o usuário terá somente que reconhecê-lo, (uma tarefa bem mais simples do ponto de vista cognitivo). E, na atividade 2, ele somente terá de preencher os campos da *template* selecionada. Assim, estas atividades fazem uso das técnicas

empregadas no paradigma de programação paramétrica para facilitar o desvelamento da *EUPL* ao usuário.

Todavia, para garantir a realização de extensões monotônicas e evitar a introdução de textos na *EUPL* que não apresente reflexo na *UIL*, o ambiente deverá auxiliar o usuário no preenchimento dos campos de uma *template*, por meio da pré-seleção dos elementos adequados a cada campo. É importante observar que os objetos que preencherão estes campos poderão ser outros comandos, como ocorre nas linguagens de programação convencionais. Assim sendo, nestas atividades, a comunicação do usuário com o software estará centrada na função metalingüística da *EUPL*, enquanto ele estiver selecionando a seqüência de comandos necessários à descrição da extensão, e na função referencial da *UIL*, enquanto ele estiver especificando as entidades a serem usadas no preenchimento dos campos das *templates*.

A atividade 3 deste processo é opcional e somente será necessária caso o usuário precise informar resultados intermediários e/ou finais dos comandos realizados pela ação. Os diálogos criados por esta atividade obedecerão ao mesmo processo descrito na etapa 5 do processo de criação de uma nova ação. No entanto, neste caso, as entidades empregadas na seleção dos *widgets* dirão respeito aos dados a serem informados na *UIL*. Devido à utilização do mesmo processo, novamente, restringimos a natureza dos diálogos criados e com isso mantemos o Contínuo Semiótico entre a *EUPL* e a *UILx* e também garantimos a consistência da função poética da *UIL*. Deste modo, como já ocorreu nas atividades da etapa 5 do processo de criação de uma nova ação, também nesta atividade a comunicação do usuário com o software estará relacionada às funções referencial e poética da *UIL*.

A etapa 7 completará o processo de criação de uma nova ação, definindo uma mensagem de erro a ser utilizada no caso de algum comando da ação falhar durante sua execução. Nesta etapa, a comunicação do usuário com o software estará voltada para a função expressiva da *UIL*. Esta etapa faz parte do mecanismo de manutenção do Ciclo Mínimo de Interação. Como citado anteriormente, ela foi inserida no processo em função dos resultados dos testes com a versão preliminar de uma *EUPL* baseada na linguagem-tipo para *EUP* proposta neste trabalho (apresentados no Anexo I). Deste modo, ela obrigará o usuário a definir a mensagem que concluirá o diálogo do software nos casos em que a ação não puder mais ser completada.

Com a exposição acima encerramos a descrição da criação de uma nova ação. Na etapa 1 da realização de uma extensão (Figura 30) temos, também, a possibilidade da criação de uma nova

entidade. Conforme descrito anteriormente, esta tarefa exige maior conhecimento do modelo de usabilidade do software sendo, portanto, menos utilizada pelo usuário final.

2.2. O processo de criação de uma nova entidade

O processo de criação de uma entidade, apresentado na Figura 32, descreve a estrutura de discurso necessária à descrição completa da nova entidade, sendo composto de quatro etapas:

1. A declaração de seu **nome**;
2. A declaração da **entidade** na qual ela é baseada (por meio da extensão de uma ou mais entidades-base ou da definição de um agregado de elementos de uma entidade já existente);
3. A definição da sua **representação na *UIL*** (por meio de uma tarefa própria para a edição da sua representação gráfica);
4. A definição da **estrutura** e **funcionalidade** particular a nova entidade (por meio de uma tarefa própria para a criação de novas ações e atributos); e

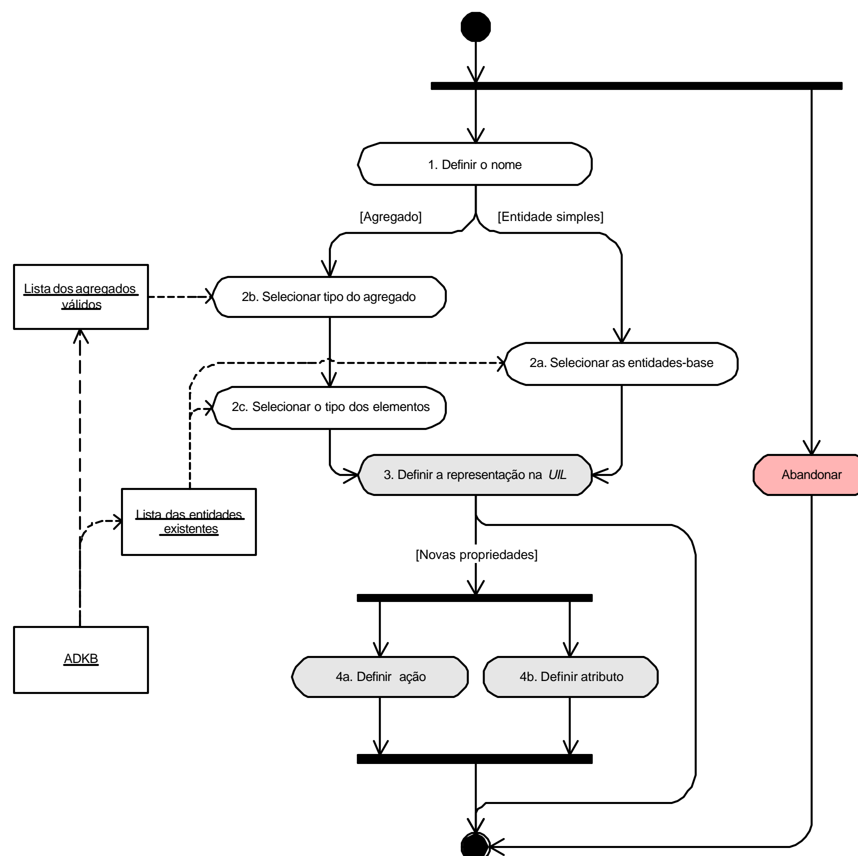

Figura 32: Descrição do processo de criação de uma nova entidade.

É importante observar que as três primeiras etapas deste processo são essenciais à existência de uma entidade, já a quarta é opcional.

Na declaração do nome da nova entidade, diferentemente do ocorrido no caso de uma ação, o nome deve ser único para não haver ambigüidade em sua referência.

É importante observar que, no Modelo Semiótico, **somente será possível gerar novas entidades a partir de uma ou mais entidades já existentes no modelo do software**. Esta restrição procura preservar mais facilmente a mensagem original do designer, contemplando também o requisito de realização de extensões monotônicas ao software, requerendo a introdução da etapa 2 no processo e o uso da função conativa da *UIL* para forçar sua efetivação por parte do usuário final. Assim, se o usuário quiser criar uma entidade que seja realmente nova ao domínio, ele deverá fazê-la a partir da raiz da hierarquia *is-a* de entidades, preservando assim, por herança, a estrutura comum das entidades.

A definição da entidade da qual a nova entidade será derivada, que ocorre na etapa 2, poderá ser realizada de duas formas mutuamente excludentes: a) por meio da definição de um conjunto de entidades base ou b) por meio da definição de um agregado de instâncias de uma entidade base (estes agregados serão listas, seqüências ou conjuntos e deverão ser estruturas predefinidas de uma *EUPL*). Assim, nestas atividades, o diálogo do usuário com o software estará centrado na função referencial da *UIL*, procurando identificar as entidades às quais ele relacionará sua nova entidade. Como esclarecido no parágrafo anterior, esta etapa está diretamente relacionada à manutenção do requisito de realização de extensões monotônicas. Por conseguinte, o ambiente de extensão deverá restringir as possibilidades de escolha do usuário às entidades do domínio da aplicação presentes na *ADKB*.

A representação da entidade na *UIL*, definida na etapa 3, faz parte do mecanismo de manutenção dos princípios de Abstração Interpretativa e do Contínuo Semiótico entre a *EUPL* e a *UILx* do software. Neste âmbito, ela obrigará o usuário a realizar a ligação entre a descrição da nova entidade na *EUPL* e sua representação na *UIL* por meio de um processo de alteração da *UIL* composto de três atividades:

- A **seleção de uma representação base** para a criação de uma nova representação;
- A **alteração da representação base** para a nova representação; e

- A **associação da nova representação** à nova entidade.

Na primeira atividade deste processo, o usuário selecionará uma representação base para produzir uma nova representação na *UIL* para a nova entidade. Esta atividade caracteriza um diálogo entre usuário e o software que estará centrado nas funções referencial e poética da *UIL*. Por atuar diretamente sobre os elementos da *UIL*, esta etapa requer um novo contexto de operação que será criado por um **agente de software editor de representações**. Assim, objetivando preservar a aparência do software, o ambiente de extensão deverá restringir as possibilidades de escolha do usuário. Isto será alcançado apresentando a ele, no editor de representações, somente o conjunto de representações formado pelas representações ligadas à categoria (entidade base/tipo do elemento do agregado) da nova entidade, definida na etapa anterior. Deste modo, o usuário terá um ponto de partida bem definido para a concepção da nova representação da entidade na *UIL*, o que possibilitará a criação de um elemento representacional que apresente alguma semelhança com as entidades das quais ela deriva e, portanto, que ofereça maior facilidade de interpretação.

Na segunda atividade deste processo, o usuário empregará o editor de representações para alterar a representação base selecionada. Nesta atividade, a comunicação do usuário com o software estará centrada na função poética da *UIL*, uma vez que ele deverá respeitar as regras de estilo válidas para o ambiente computacional em questão. Infelizmente, esta atividade não tem como ser totalmente controlada e, portanto, apesar de o usuário ter sido guiado na atividade de seleção da representação base, torna-se impossível evitar que ele crie incoerências representacionais na *UIL*[59]. Esta atividade também é necessária para marcar a distinção entre um elemento desta nova entidade e os elementos das entidades dos quais ela deriva.

A atividade 3 é automática e associará a nova representação à nova entidade que está sendo gerada, fechando, assim, a etapa de definição da representação de uma entidade na *UIL*.

Retornando ao processo de criação de entidades, a etapa 4 é opcional e definirá as estruturas e funcionalidades que são particulares à nova entidade (isto é, as características que a diferenciam das entidades das quais ela deriva). Estes elementos serão especificados por meio da adição de novas ações (atividade 4a) e/ou novos atributos (atividade 4b) à entidade. É importante observar que, no Modelo Semiótico, **todo atributo tem de ser empregado em alguma ação para ser válido**. Esta restrição garantirá a criação de modelos de domínio mínimos, facilitando a

[59] É sempre bom lembrar que a relação entre um signo e o objeto que ele representa é arbitrária.

geração de texto explicativo para a *UEL* a partir dos elementos da *ADKB*. Ela deverá ser verificada na fase de validação sintática da extensão. É interessante notar que, caso esta etapa não seja realizada (e a entidade não seja um agregado), o usuário estará criando somente um **sinônimo** — um *alias* —para alguma entidade já existente na *ADKB*, o que pode ser útil para corrigir problemas referentes à linguagem empregada pelo designer do software. Um uso típico é a substituição do nome escolhido pelo designer de software para uma entidade por outro que é de uso mais comum para o usuário no domínio em questão. Assim, por exemplo, no domínio do *E-mailer* apresentado na Figura 28, no Capítulo 3, o usuário poderia trocar o nome da entidade *Folder* por *Directory* empregando a extensão:

```
Directory is a folder.
```

Sem adicionar uma nova propriedade a esta nova entidade.

A tarefa de adição de ações à entidade emprega um processo semelhante ao anteriormente descrito para a criação de uma única ação, apresentado na Figura 31. Contudo, neste caso, a entidade à qual a ação será associada já é conhecida, eliminando a necessidade da etapa 3 daquele processo. Além disso, o usuário poderá criar mais de uma ação por vez, retornando ao início do processo sempre que quiser. A tarefa de criação de atributos é realizada pelo processo apresentado na Figura 33, que é composto de cinco atividades:

1. A declaração de seu **nome**;
2. A declaração de sua **faixa de valores**;
3. A declaração de seu **valor *default***;
4. A definição de sua **forma de cálculo** (por meio de tarefa própria para definição do procedimento de cálculo); e
5. A definição de sua **forma de interação com a *UIL*** (por meio de tarefa própria para definição das condições que levam o atributo a alterar a representação da entidade na *UIL*).

As duas primeiras atividades são obrigatórias para a definição do atributo, as demais são opcionais e definem características especiais de um atributo.

O nome do atributo, definido na atividade 1, também deverá ser único dentro da entidade para evitar ambigüidades em sua referência. A definição da faixa de valores do atributo, realizada na atividade 2, pode ser feita de duas formas mutuamente excludentes, declarando:

1. A entidade à qual o atributo pertence (indiretamente declarando sua faixa de valores); ou
2. Diretamente uma enumeração dos valores válidos para o atributo.

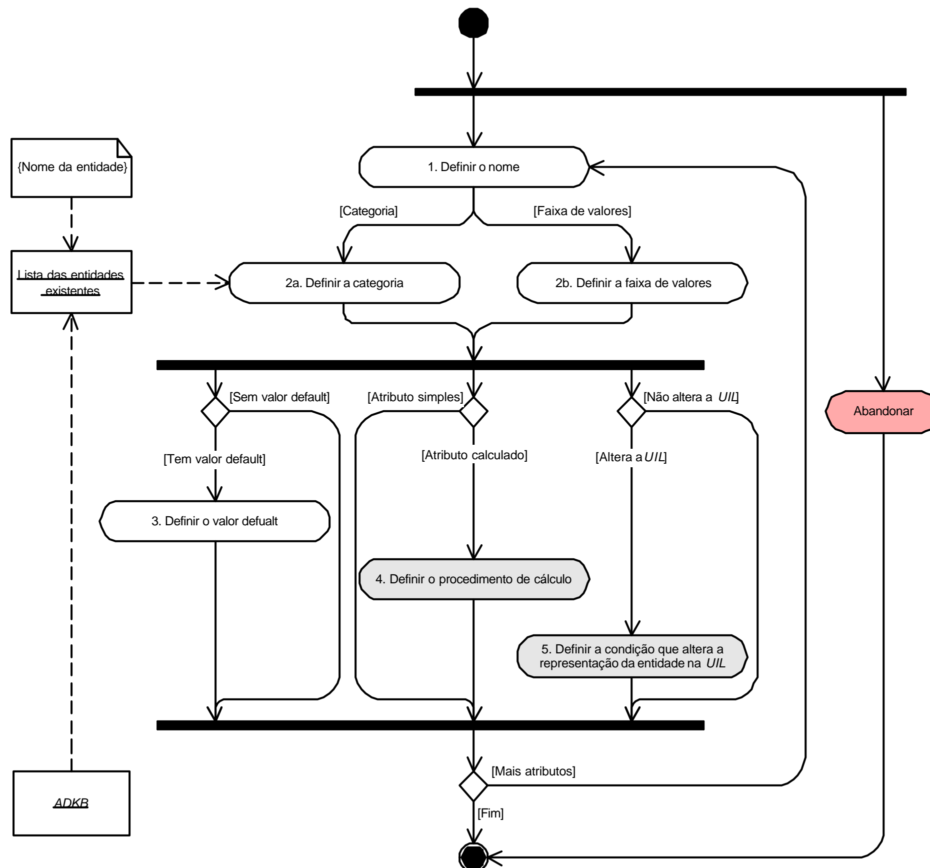

Figura 33: Descrição do processo de definição de atributos para as entidades.

Nesta atividade, o diálogo do usuário com o software deverá estar centrado na função referencial da *UIL* e, portanto, será necessário, na opção 1, que o ambiente de extensão o auxilie apresentando as entidades da *ADKB* possíveis de serem usadas. Na opção 2, como ocorreu na criação de uma representação para a entidade na *UIL*, não será possível controlar os nomes dos valores declarados pelo usuário.

A atividade 3, que é opcional, permitirá ao usuário definir um valor *default* para o atributo, caso ele exista. Deste modo, o diálogo entre o usuário e o software, nesta atividade, também se concentra na função referencial da *UIL*. A razão para esta atividade ser opcional está no fato de que nem sempre está claro para o usuário qual deverá ser o valor que melhor representa o *default* do um atributo. Assim, em vez de obrigá-lo a definir um valor, que poderá não ter sentido para ele, é mais interessante permitir a ele que o defina somente se for realmente necessário. Esta atividade é

totalmente dependente da atividade 2, visto que o valor do novo atributo deverá estar dentro da sua faixa de valores válidos. Além disso, o fato desta atividade ser opcional traz conseqüências para a estrutura de uma *EUPL*, uma vez que será necessário definir a existência de um valor indefinido que será atribuído aos atributos quando eles são criados, de modo a evitar cálculos errados.

É importante observar que, no Modelo Semiótico, um atributo pode ter seu valor definido (caso em que este valor é atribuído pelo usuário ou pelo software) ou computado (caso em que seu valor é calculado pelo software sempre que for necessário). Assim, a atividade 4, que também é opcional, permitirá ao usuário definir um procedimento de cálculo para o valor do atributo caso este seja do tipo computado. A definição do procedimento de cálculo empregará um processo semelhante ao anteriormente descrito para a definição de comandos, no processo de criação de uma nova ação. Todavia, neste caso, não haverá a necessidade da atividade de definição de diálogos de saída de dados, uma vez que o resultado dos cálculos efetivados pelos comandos será conferido diretamente ao atributo.

A atividade 5 permitirá ao usuário definir uma condição sobre o valor do atributo que, uma vez satisfeita, provocará uma alteração na representação da entidade na *UIL*. Neste caso, a comunicação do usuário com o software estará voltada para a função expressiva da *UIL*, procurando passar ao usuário o valor das alterações por ele realizadas. Esta atividade é mais uma atividade opcional neste processo, já que nem todos os atributos influem na representação (ou seja, na forma de expressão) de uma entidade na *UIL*. A complexidade da condição dependerá do conhecimento lingüístico e computacional que o usuário detém. Deste modo, o usuário deverá ser auxiliado nesta tarefa por um agente de software específico para a aquisição destas condições. Ao definir a condição de alteração da representação da entidade, o usuário também deverá definir a nova representação que deverá ser usada. Com este propósito, ele utilizará o mesmo processo empregado na criação de uma representação para a entidade na *UIL*, descrito na etapa 3 do processo de criação de entidades. Assim, esta atividade também está relacionada ao requisito de manutenção do Contínuo Semiótico entre a *EUPL* e a *UILx*.

Com esta exposição, encerramos a descrição da criação de uma entidade. Na etapa 1 da realização de uma extensão ainda temos a possibilidade da criação de uma nova relação. Conforme descrito anteriormente, esta tarefa exige um alto grau de conhecimento do modelo de usabilidade do software sendo, portanto, também pouco utilizada pelo usuário final.

2.3. O processo de criação de uma relação

No processo de criação de uma nova relação entre entidades, o usuário definirá quais entidades apresentam um inter-relacionamento significante para o modelo do software. Tal relacionamento pode ser utilizado por mecanismos internos do software para cumprir diversas tarefas como, por exemplo, realizar inferências, empregando técnicas de IA, que possibilitam um tratamento lingüístico mais avançado para uma *EUPL*. Na definição desta inter-relação, o usuário fará uso de um grupo de **relações predefinidas** pelo designer do software. É importante observar que não é possível, nem interessante, deixar livre o tipo de relações que podem ser realizadas pelo usuário, pois toda relação deverá ser acompanhada de um mecanismo de inferência que faça uso dela, caso contrário ela não seria útil ao modelo do domínio.

Conforme discutido no Capítulo 5, existem duas relações predefinidas nas *EUPLs* derivadas da linguagem-tipo para *EUP* proposta neste trabalho (a saber, as relações de herança e de todo-parte) que são relações estruturais sustentadas diretamente pela sintaxe de tais *EUPLs*. A hierarquia formada pelas entidades que apresentam relações de herança servirá, como nos sistemas orientados a objetos, para determinar se uma entidade detém certa ação ou não e para manter uma estrutura interna mínima para as entidades. A hierarquia formada pelas entidades que apresentam relações de todo-parte será primordial para o **mecanismo de resolução de anáforas e elipses** (instrumentos lingüísticos que deverão fazer parte de uma *EUPL* instanciada a partir da linguagem-tipo para *EUP* proposta neste trabalho, conforme visto no Capítulo 5).

Entretanto, caso a *EUPL* do software opere também com o **mecanismo de resolução de metáforas e metonímias** (outros instrumentos lingüísticos que podem ser empregados em uma *EUPL* para enriquecer seu poder de expressão), o conjunto de relações predefinidas aumentará consideravelmente, uma vez que será necessário marcar na *ADKB* as relações que poderão ser empregadas para este objetivo. Um exemplo de um conjunto de relações úteis para este fim pode ser encontrado em [BARBOSA '99]. Porém, estas relações são dependentes dos mecanismos de inferência disponibilizados e, portanto, não serão inseridas diretamente na linguagem-tipo para *EUP* proposta. O processo de criação de relações contém 3 atividades:

- A escolha da **primeira entidade**;
- A escolha do **tipo de relação** (dentre as relações predefinidas pelo designer); e
- A escolha da **segunda entidade**.

Assim, este modelo somente apresenta relações binárias entre as entidades, visto elas cobrirem uma percentagem razoável das relações que é necessário modelar em domínios computacionais. Deste modo, nesta tarefa, a comunicação entre o usuário e o software estará centrada nas funções metalingüística e referencial da *UIL* e, portanto, o ambiente de extensão deverá auxiliá-lo por meio da apresentação das relações predefinidas na *ADKB* e das entidades que podem participar destas relações.

2.4. O processo de modificação de uma extensão

Na etapa de especificação da extensão temos também a possibilidade de modificar ações, entidades, atributos e representações de entidades, diálogos de aquisição ou apresentação de dados e relações entre entidades presentes no modelo da aplicação. A tarefa de modificação de cada um destes elementos deve possibilitar ao usuário alterar cada uma das etapas dos seus respectivos processos de criação. No entanto, existe uma restrição, ilustrada na Figura 34, para o caso particular da modificação de uma entidade, que se aplica a todos os processos de modificação. Esta restrição está relacionada à manutenção do requisito de realização de extensões monotônicas ao software.

Todo elemento que se encontra na *ADKB* deve pertencer ao design original do software ou a uma extensão do usuário. Por conseguinte, no processo de modificação de uma extensão, o ambiente deverá eliminar, automaticamente, a possibilidade de alteração dos elementos pertencentes ao design original do software, de modo a também respeitar o requisito de realização de extensões monotônicas. Para este fim, ele deve apresentar ao usuário somente os elementos por ele criados (ou seja, elementos que são resultado de suas extensões).

No entanto, este não é o único problema para a modificação de uma extensão. Após o usuário ter selecionado o elemento que deseja alterar (no caso da Figura 34, uma entidade), é necessário que o mecanismo de extensão verifique a existência de dependências entre este elemento e os demais elementos da *ADKB*. Estas dependências deverão ser apresentadas ao usuário e ele deverá ser alertado, por meio da função conativa da *UIL*, para o fato de que não poderá alterar o elemento escolhido enquanto as dependências não forem resolvidas, visto que isto poderá acarretar o mau funcionamento dos elementos dependentes.

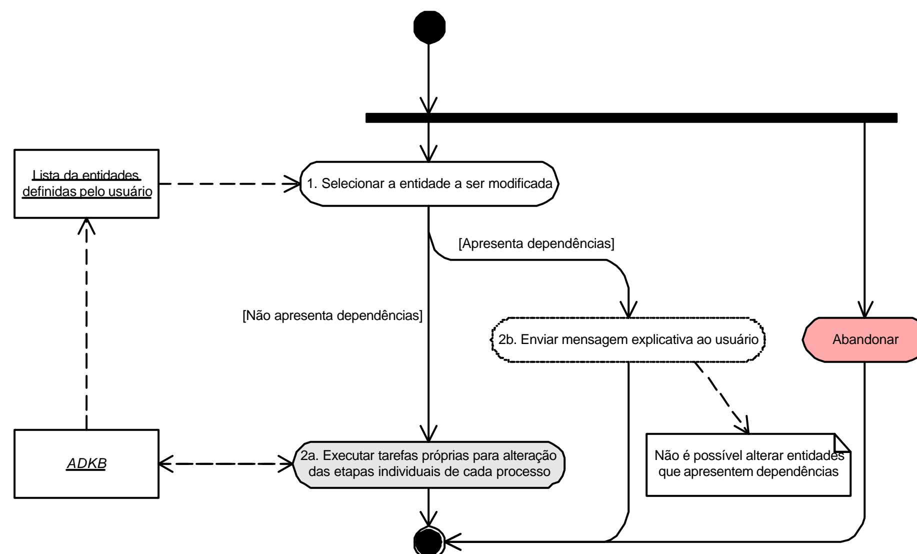

Figura 34: Processo abstrato de modificação de uma entidade do modelo do software.

É importante observar que não é aconselhável fazer a eliminação automática dos elementos que apresentam dependência entre si, pois a real intenção do usuário pode ser a de alterar tal elemento independentemente do esforço necessário para alterar suas dependências. Assim sendo, esta restrição procura impedir que uma modificação realizada por um usuário altere a semântica de outros elementos presentes na *ADKB* e, com isto, destrua o modelo de usabilidade da aplicação desenvolvido até então, o que também acarretaria a violação do requisito de realização de extensões monotônicas ao software.

2.5. O processo de revogação de uma extensão

A última tarefa possível de ser realizada na etapa de especificação da extensão é a revogação de ações, entidades, atributos ou relações presentes no modelo da aplicação. A palavra revogação, conforme dissemos antes, é tomada aqui com licença de expressão, pois ela não é de fato uma extensão visto provocar o encolhimento da *ADKB*. No entanto, ela pode ser vista como o caso extremo da tarefa de modificação descrita na seção anterior, isto é, quando a modificação implica a anulação do elemento modificado. Deste modo, ela apresenta as mesmas restrições já discutidas para aquela tarefa. O processo de revogação difere do processo de modificação somente no conjunto de tarefas a serem realizadas após o teste de dependência, as quais são dependentes do tipo de elemento revogado da *ADKB*.

3. A etapa de validação sintática

Uma vez especificada a extensão, será necessário verificar a consistência da descrição feita pelo usuário para averiguar se o texto por ele criado é, de fato, um texto válido na *EUPL* utilizada no ambiente de extensão. É essencial observar que o processo de especificação da extensão, adotado no Modelo Semiótico da tarefa de *EUP*, restringe grande parte das possibilidades de introdução de erros no texto. Todavia, ainda existe uma situação em que é possível ao usuário introduzir erros. Ela ocorrerá quando, na tarefa de criação de uma entidade, o usuário criar atributos que não sejam empregados em alguma ação da entidade que ele esteja criando. Felizmente, os erros resultantes desta situação podem ser capturados analisando a sintaxe do texto da extensão.

Assim, o processo de verificação de consistência do texto da extensão evita que estes atributos sejam adicionados à *ADKB*, uma vez que eles poderiam interferir no mecanismo de explicação da *UEL*. Caso seja identificado este tipo de erro, o ambiente de extensão, por meio da função conativa da *UIL*, chamará a atenção do usuário para o problema, procurando oferecer auxílio à sua resolução através de um agente de software, projetado para apoiá-lo na tarefa de depuração do texto da extensão, agente este que chamará o editor de extensões na posição em que o atributo foi definido para que o usuário possa identificar o seu uso ou cancelar sua declaração.

É interessante observar que existem duas outras situações que poderiam ser consideradas como geradoras de erros no Modelo Semiótico, são elas: 1) o emprego da sintaxe errada no uso de ações internas a outras ações e 2) a definição de entidades que não apresentem ações próprias. A primeira situação será eliminada por meio do uso de um editor de comandos dirigido por sintaxe (ou com o uso de *templates*, conforme citado anteriormente) que possa consultar a *ADKB* e apresentar a sintaxe de uso de cada ação ao usuário[60]. A segunda situação apresenta três casos distintos, são eles:

1. Quando a entidade que está sendo definida tem somente uma entidade base, caso em que a situação será, por definição, considerada válida neste modelo, visto que ela definirá um sinônimo para a entidade do usuário;

---

[60] Este tipo de editor é empregado na maioria dos ambientes de desenvolvimento comerciais atuais.

2. Quando a entidade que está sendo definida tem mais de uma entidade base, caso em que ela será considerada inválida, uma vez que não é lingüisticamente correto criar um sinônimo nesta situação; e
3. Quando a entidade que está sendo definida é um agregado, caso em que ela não criará sinônimo, mas que será considerada válida, pois o agregado já contém ações próprias de caminhamento que definem sua funcionalidade.

4. A etapa de validação semântica

Após a verificação de consistência sintática do texto, teremos uma extensão sintaticamente correta em que todos os elementos do seu texto são realmente utilizados. Todavia, será necessário testá-la para verificar se ela realmente realiza a tarefa desejada, isto é, se ela está semanticamente correta. Para aumentar as chances de que uma extensão má formada não destrua o trabalho do usuário, a tarefa de teste das extensões deverá ocorrer em um contexto diferente do ambiente de uso ou de extensão — ou seja, ela deverá ocorrer em um **ambiente de teste**. O processo de teste da extensão, apresentado na Figura 35, é composto de quatro etapas:

1. A **criação do ambiente de teste**;
2. A **seleção do passo** a ser testado;
3. A **execução e verificação da validade do passo** (com a apresentação do resultado ao usuário);
4. A **correção do passo**, pelo usuário, se necessário, e o *reinício* do teste.

A geração de uma cópia do ambiente de trabalho atual do usuário, realizada na etapa 1, deverá ser automática e é essencial ao processo de teste, conforme esclarecido no parágrafo anterior. O mesmo ocorrerá com a etapa 2, pois o ambiente de teste sempre saberá qual o próximo passo a ser testado em uma extensão.

Na etapa 3, o passo selecionado será executado. Esta atividade poderá requerer a intervenção do usuário para a realização de uma ação estipulada pelo passo que está sendo testado. Nesta etapa, é essencial para o usuário que o software faça uso extensivo da função fática da *UIL* para comunicar seu estado, uma vez que a probabilidade de que um erro bloqueie seu funcionamento é muito grande. Após a execução do passo, existem dois caminhos possíveis para o

processo. No primeiro, o passo executado está correto (segundo a análise do usuário) e o processo retornará à etapa 2 para selecionar o próximo passo a ser testado (contanto que não seja este o último passo da extensão). No segundo, o passo executado apresenta algum erro (segundo a análise do usuário) e o processo passará para a etapa 4a, na qual o ambiente fará uso da função fática para levar o usuário a corrigir o erro. Esta etapa fará uso do mesmo agente de software para depuração de extensões empregado na etapa de validação sintática do processo global de realização de uma extensão, levando o usuário à posição exata do passo na sua extensão.

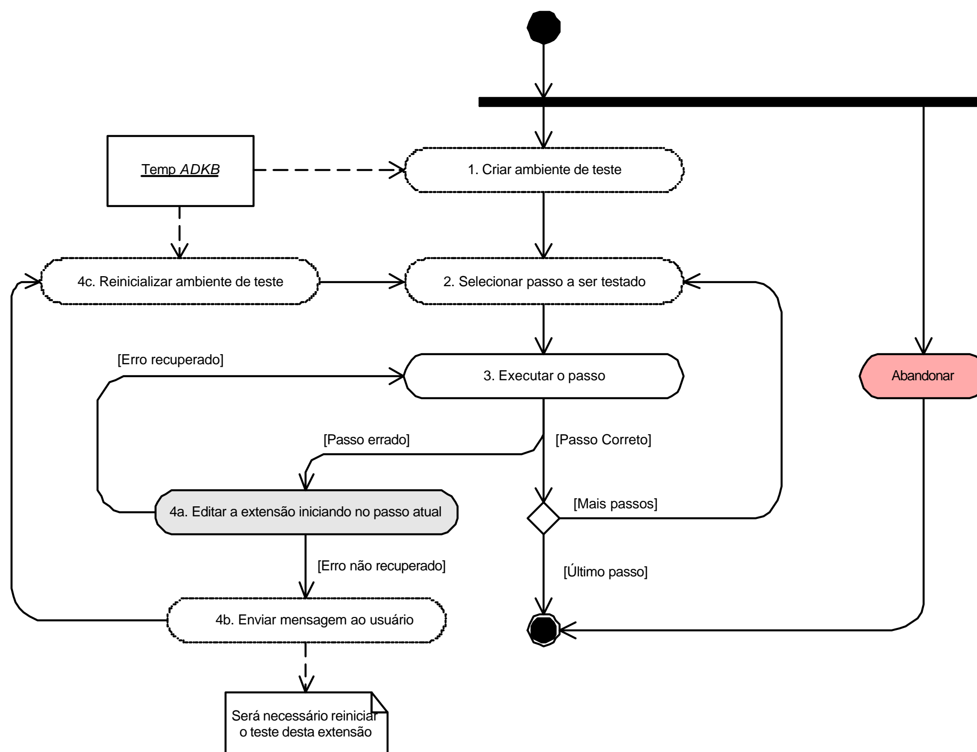

Figura 35: Processo de teste da validade semântica de uma extensão ao software.

O erro identificado pode afetar somente o passo em questão, caso em que é possível continuar o teste a partir da re-execução do passo corrigido, levando o processo a retornar a etapa 3. Ele poderá afetar também mais de um passo da extensão, caso em que haverá a necessidade de se reiniciar o teste da extensão desde o início. Neste caso, o ambiente de extensão deverá fazer uso da função expressiva da *UIL* para alertar o usuário sobre a necessidade do *reinício* do teste, o que é realizado na etapa 4b. A retomada do teste é feita após a re-inicialização do ambiente de teste, na etapa 4c.

Somente após a operação estar testada e funcionando é que ela será armazenada de forma definitiva na *ADKB*. Assim, podemos garantir que a *ADKB* estará necessariamente atualizada e que a nova extensão poderá ser empregada na geração de explicações pela *UEL*.

5. O abandono da tarefa de *EUP*

Conforme citado anteriormente, no Modelo Semiótico o usuário sempre poderá abandonar a atividade que está realizando. Contudo, ao resolver abandonar seu trabalho, ele poderá estar somente interessado em postergá-lo. Basicamente, o usuário poderá escolher entre abandonar totalmente sua extensão, caso em que as alterações intermediárias feitas à *ADKB* serão eliminadas, ou poderá escolher salvar estas alterações para continuar seu trabalho em outro momento, caso em que será criada uma cópia temporária da *ADKB* (somente com as alterações intermediárias). Na segunda possibilidade, o ambiente de extensão deverá fazer uso da função expressiva da *UIL* para alertar ao usuário que alguns dos nomes que ele empregou até o momento, na criação da extensão interrompida, poderão não estar disponíveis até a conclusão da extensão.

6. Comentários finais à organização do processo

O ambiente de extensão tem papel preponderante na usabilidade de um mecanismo de extensão. Tanto é assim que a maioria dos paradigmas apresentados no Capítulo 2 apresentam um ambiente à parte para a tarefa de criação de extensões. Um ambiente de extensão é na verdade uma interface para o processo que rege a realização das atividades da tarefa de *EUP* que determinado modelo segue. Ele é responsável por garantir que o processo seja realizado da melhor forma possível. Para isto, ele deve disponibilizar mecanismos que apóiem o usuário na realização deste processo.

Conforme citado anteriormente, o ambiente de extensão deverá disponibilizar para o usuário um agente de software, na forma de uma Agenda, que descreva as tarefas a serem realizadas para que a extensão esteja completamente definida. Porém, o suporte à construção da extensão pode ir bem mais longe. Um ponto crítico em todos os ambientes de extensão que operam com linguagem textual está na redação da própria extensão. Dependendo do poder de expressão da linguagem empregada a quantidade de texto a ser escrita pelo usuário é muito elevada. Além disso, por ser esta uma atividade esporádica, o usuário normalmente esquece a gramática da linguagem e, principalmente, o nome das entidades que podem ser manipuladas e sua composição.

O Modelo Semiótico procurou resolver, diretamente na estrutura da linguagem-tipo para *EUP*, alguns dos elementos de dificuldade levantados no parágrafo anterior. O poder de expressão de uma *EUPL* que siga esta linguagem-tipo foi elevado para próximo do nível da linguagem natural. Deste modo, a gramática da linguagem foi simplificada para uma gramática parcialmente conhecida do usuário. Considerando que o conjunto de regras gramaticais de uma *EUPL* deste tipo é bastante reduzido, é possível fazer uso, no ambiente de extensão, de um conjunto de *templates* para as regras gramaticais ampliando, assim, a facilidade de uso de sua gramática. Deste modo, estaremos trocando a tarefa cognitiva do usuário do processo de lembrança para o processo de reconhecimento, que é mais eficiente, conforme nos mostram os estudos da psicologia cognitiva [PREECE '94].

O uso de *templates* para as regras gramaticais da *EUPL* não somente facilita o reconhecimento destas regras pelos usuários mas, também, evita que este usuário escreva textos inválidos na *EUPL*. Porém, ainda resta o problema do esquecimento dos nomes das entidades manipuláveis. Contudo, o uso de *templates* abre as portas para o emprego de elementos de interface (como, por exemplo, *combobox*, *listbox*, entre outros) que podem apresentar ao usuário os nomes válidos em cada campo da *template* após fazer uma consulta à *ADKB*, conforme previsto no processo do Modelo Semiótico, apresentado nas seções anteriores deste capítulo. Isto facilita a escolha das entidades e ações a serem usadas naquelas regras pelo usuário reduzindo, portanto, ainda mais a carga cognitiva de lembrança destes nomes.

O *layout* de um ambiente de extensão pode ser muito variável, mas em um mecanismo que implemente o Modelo Semiótico proposto, de alguma forma, ele deverá apresentar, conforme ilustrado na Figura 36, pelo menos as quatro seguintes áreas de interação:

- Uma **área de Agenda**, onde será indicado ao usuário quais tarefas faltam ser realizadas para a construção da extensão;
- Uma **área de *Templates***, onde serão apresentadas ao usuário as regras gramaticais que podem ser empregadas na situação em questão;
- Uma **área de Texto**, onde o usuário entrará o texto da extensão em si; e
- Uma **área de Explicação**, onde serão apresentadas ao usuário as mensagens de erro e as explicações sobre suas dúvidas.

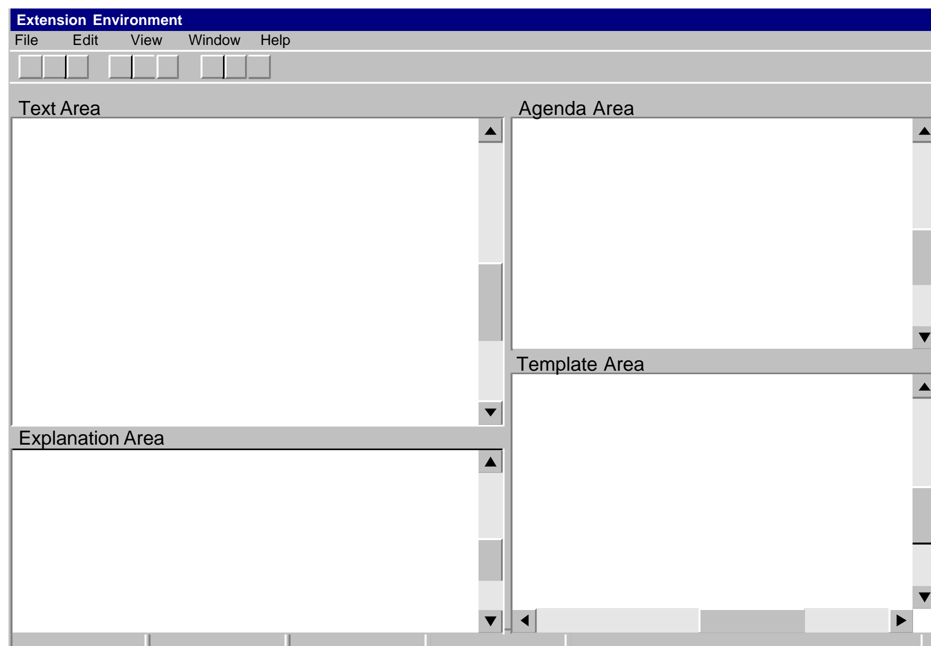

Figura 36: Uma configuração para um ambiente de extensão para o Modelo Semiótico proposto.

No entanto, é importante lembrarmos que esta é somente uma das possíveis configurações para este ambiente.

Com isto, fechamos a descrição do Modelo Semiótico para a tarefa de *EUP* proposto neste trabalho. Nela apresentamos um modelo teórico para a tarefa de *EUP* baseado na Engenharia Semiótica, delineamos a estrutura de uma linguagem-tipo para *EUP* que respeita os princípios levantados neste modelo, descrevemos a organização de uma arquitetura de software que dê apoio à manutenção dos requisitos levantados neste modelo teórico e descrevemos um processo para a tarefa de *EUP*. No próximo capítulo apresentaremos uma análise dos resultados obtidos neste trabalho e algumas idéias para sua continuação.

Capítulo 7

CONCLUSÕES

Neste capítulo, apresentamos nossas principais conclusões sobre este trabalho, para tal comparamos nossa abordagem com os paradigmas atuais de programação por usuários finais discutidos no Capítulo 2 procurando apresentar suas vantagens e limitações. Além disso, relacionamos nossas contribuições para a área de *EUP*. Por fim, apresentamos algumas idéias para a continuação da pesquisa desenvolvida neste trabalho e para a sua aplicação a outras áreas de conhecimento.

Um elemento crucial para que possamos comparar as técnicas empregadas na área de *EUP* é a existência de modelos teóricos que fundamentem a arquitetura dos mecanismos tecnológicos que as implementam. A atual ausência, na literatura, deste tipo de modelo nos levou a classificar os mecanismos de extensão por usuários finais, encontrados na literatura acadêmica e nas aplicações de software desenvolvidas pela indústria, baseando-nos nos paradigmas de computação por eles empregados. Como resultado, identificamos a existência de três paradigmas de programação atualmente empregados nestes mecanismos: o paramétrico, o imitativo e o descritivo.

Nossa análise destes mecanismos também nos mostrou que todos eles sofrem de um problema genérico com respeito aos critérios de usabilidade levantados por Adler e Winograd, uma vez que a maioria não apresenta um bom auxílio à aprendizagem de seu modelo de usabilidade e extensibilidade. Este problema é um reflexo direto da inexistência de uma linguagem de explicação que seja semioticamente contínua com as linguagens de interface e de extensão presentes nas suas implementações. O sistema de *help* presente nestes mecanismos é somente uma parte de uma linguagem de explicação e, normalmente, não leva em conta os aspectos pragmáticos (de uso) de uma aplicação. Como resultado, ele gera uma "explicação" que, na maioria das vezes, não é suficiente para esclarecer as dúvidas dos usuários.

Assim sendo, a contribuição mais significativa deste trabalho é a descrição de um modelo teórico para a tarefa de *EUP* que leva em conta os aspectos comunicativos, representacionais e de aprendizado levantados por Adler e Winograd [ADLER '92], conforme foi descrito nos Capítulos 3, 5 e 6 deste trabalho. Nosso modelo baseia-se na Engenharia Semiótica [DE SOUZA '93] e no Modelo de Comunicação Verbal de Jakobson [JAKOBSON '60] para propor um processo para a tarefa de realização de extensões a aplicações e uma linguagem-tipo para *EUP*. Estes dois elementos apresentam uma forte integração proporcionada pelos mecanismos de manutenção dos princípios da Abstração Interpretativa e do Contínuo Semiótico, que são a base de nosso modelo teórico.

O desenvolvimento deste modelo resultou ainda na definição de uma nova arquitetura de software que faz uso de uma base de conhecimento do design da aplicação — a *ADKB* — para coordenar a integração acima descrita. Esta base de conhecimento é composta pelo modelo do domínio da aplicação e pelo modelo do software que, entre outras coisas, representa as regras de design empregadas pelo designer de software na criação da aplicação. É importante observar que a *ADKB* precisa conter somente os elementos que são visíveis aos usuários finais, ou seja, ela representa o modelo de usabilidade do software apreendido por estes usuários. Esta composição é semelhante à proposta por Liberman [LIBERMAN '98] em seus estudos sobre a introdução de agentes de software em aplicações convencionais. Ele nos mostra que, para que os agentes possam operar de forma satisfatória, é necessário que eles tenham um "modelo mental" da aplicação equivalente ao modelo de usabilidade formado pelos usuários. Deste modo, tal modelo não tem de conter todos os elementos da arquitetura da aplicação.

É relevante observar que a arquitetura proposta neste trabalho é centrada num mecanismo de explicação que gera uma linguagem de explicação[61] semioticamente contínua com a *UIL* e com a *EUPL* da aplicação. Portanto, uma vantagem expressiva da existência de uma base de conhecimento com esta estrutura ligada à aplicação é o suporte aos aspectos pragmáticos do domínio em questão na geração de textos explicativos para as dúvidas dos usuários. Deste modo, nossa proposta apresenta uma resposta teoricamente satisfatória às necessidades de explicação e aprendizagem dos usuários finais conforme as propostas de Adler e Winograd, o que não ocorre nos paradigmas atuais de programação empregados em *EUP*.

Como vimos no Capítulo 2, o paradigma de programação paramétrica opera pela composição simples de elementos primitivos de um conjunto fechado para gerar uma extensão à aplicação, sendo a ordem de escolha destes parâmetros irrelevante. Este paradigma é adequado para uma série de extensões atemporais e independentes de uma cadeia causal mediata. São exemplos destas extensões a adaptação da aparência de uma aplicação, a configuração de opções e o uso de moldes.

Do ponto de vista cognitivo, podemos observar que este paradigma apresenta um problema bastante significativo quando o número de opções é grande, pois há também um grande número de combinações possíveis entre elas, impondo uma carga cognitiva considerável ao usuário que precisa decidir o que fazer. Este problema é mais facilmente sentido nos mecanismos de configuração de opções da aplicação, mas também pode ocorrer nos mecanismos de adaptação da aparência de uma aplicação.

Se considerarmos os aspectos semióticos, nosso modelo teórico nos possibilita prever dois graves problemas relacionados a este paradigma. O primeiro, e provavelmente o mais sério deles, cria a possibilidade de violação do princípio de Abstração Interpretativa. Este problema ocorrerá a partir do momento que um usuário necessite de informação de um nível de linguagem inferior ao da *UIL* da aplicação para determinar a forma correta de uso de um parâmetro. Isto é, o grau de informação sobre o funcionamento da aplicação requerido para que um usuário possa selecionar, consciente e corretamente, um parâmetro é maior que o disponível na *UIL*. Este problema é resultante do fato de que a informação requerida para a utilização correta de um parâmetro faz parte da metalinguagem da aplicação. Como o grão de informação da metalinguagem e da *UIL* não

---

[61] Este aspecto não foi diretamente tratado neste trabalho, mas ajuda a amenizar o problema de aprendizagem descrito por Adler e Winograd [Adler '92].

é o mesmo, torna-se impossível prever na fase de design do software uma forma de fazer com que o nível de parametrização, na parte extensível da aplicação (a *UILx*), se coadune com a informação que o usuário detém ou pode adquirir da aplicação a partir do modelo de usabilidade criado pelo contato com sua *UIL*. Neste ponto, o usuário terá de recorrer a um conhecimento do nível de implementação da aplicação que não está disponível na *UIL* e, portanto, ocorre uma quebra do princípio de Abstração Interpretativa que resultará na impossibilidade de uso correto do mecanismo de extensão. Este problema é mais comum nos mecanismos de configuração de opções da aplicação.

O segundo problema que podemos prever para este paradigma é resultante da possibilidade da violação do princípio do Contínuo Semiótico. Este problema ocorrerá sempre que for possível ao usuário alterar a mensagem original do designer, ou seja, quando ele puder alterar a estrutura original de funcionamento da aplicação. Isto acontecerá sempre que um mecanismo de extensão permitir a modificação das relações que definem a semântica de um elemento original da *UIL* (as quais são determinadas pela associação entre este elemento e uma funcionalidade do software) como, por exemplo, quando o mecanismo de adaptação da aparência da aplicação permitir ao usuário alterar a associação de um ícone de uma barra de ferramentas para o acionamento de uma funcionalidade diferente daquela para qual o designer o havia projetado. Esta alteração trabalha contra o aprendizado do modelo de usabilidade já adquirido pelo usuário, resultando na dificuldade de uso da funcionalidade alterada e de compreensão do próprio modelo de usabilidade da aplicação.

Assim sendo, podemos ver que o Modelo Semiótico proposto neste trabalho é mais preciso e expressivo que o paradigma de programação paramétrica, uma vez que ele leva em conta a ordem em que as tarefas são realizadas e não apresenta os problemas de inadequação do nível de informação disponível para o seu uso, nem da destruição da estrutura original da aplicação.

Todavia, é importante observar que as técnicas empregadas em ambos não se contrapõem. Assim, é possível integrá-las em uma única aplicação desde que ambas respeitem os princípios da Engenharia Semiótica descritos neste trabalho. Um exemplo bastante simples desta integração seria o emprego da *ADKB* que suporta o Modelo Semiótico para introduzir uma linguagem de explicação nos mecanismos de configuração de opções da aplicação, com a concomitante adequação do nível de parametrização. Deste modo, seria possível evitar a violação do princípio de Abstração Interpretativa e empregar a programação paramétrica nas tarefas que para as quais ela apresenta bons resultados.

Porém, é possível ir mais longe, fazendo uso das habilidades cognitivas do mecanismo de moldes que, como foi descrito no Capítulo 2, agrupam padrões comuns de interação e os associam a um elemento da interface. Possibilitando a captura de padrões conhecidos aos usuários, os moldes podem tirar proveito da conhecida superioridade do fenômeno cognitivo de reconhecimento sobre o de lembrança [PREECE '94], facilitando, assim, as tarefas de identificação e aprendizado da função executada por cada extensão. Esta capacidade também pode ser empregada para promover a aquisição da sintaxe da *EUPL* empregada no mecanismo de extensão, conforme discutido no capítulo 6. Caberá aos mecanismos de manutenção do Contínuo Semiótico garantir que os moldes criados pelo usuário contenham somente elementos que respeitem os princípios da Engenharia Semiótica. Estes mecanismos, aliados a uma *UEL* baseada na *ADKB*, possibilitam aumentar a compreensão que o usuário terá da lógica de funcionamento da aplicação e, portanto, ampliar sua aplicabilidade e usabilidade.

No Capítulo 2 vimos que, comparado à programação paramétrica, o paradigma de programação imitativa amplia o escopo de extensões possíveis, levando em conta a ordem na qual os elementos são dispostos. Do ponto de vista lingüístico, podemos dizer que ele passa do nível léxico (correspondente à programação paramétrica, em que se alteram os parâmetros de significado de determinados componentes funcionais ou interativos da aplicação) para o sintático (correspondente à programação imitativa, em que se pode tornar significativa uma alteração da ordem dos elementos numa estrutura qualquer). Este aumento no poder de expressão representa um custo adicional para o usuário que agora terá de aprender quais as seqüências válidas ou, na dimensão lingüística, quais as estruturas sintaticamente válidas em uma extensão de software. São exemplos deste paradigma de programação a criação de macros e a programação por demonstração.

Apesar de ter ampliado o poder expressivo da *EUPL* (por considerar extensões temporais com uma cadeia causal) e de ser cognitivamente atraente (por empregar como sua *EUPL* a própria *UIL* da aplicação, que é um elemento já conhecido do usuário), este paradigma apresenta sérios problemas no que diz respeito à expressão de condicionais, iterações e variáveis. Estes problemas estão relacionados ao fato destes mecanismos somente criarem réplicas das seqüências de ações (processos) especificadas e, além disso, destas réplicas capturarem elementos arbitrários do contexto de uso da aplicação, conforme discutido no Capítulo 2.

Considerando os aspectos semióticos deste paradigma, nosso modelo teórico nos possibilita prever a possível ocorrência de problemas com os mecanismos que realizam este paradigma toda vez que um elemento comum da *UIL* (um elemento que não pertença à metalinguagem de extensão) não possa, por razões arbitrárias, ser empregado na criação de extensões. Este fato, que ocorre em algumas aplicações comerciais que empregam mecanismos deste paradigma, gera uma violação do princípio do Contínuo Semiótico e concorre para dificultar a compreensão do funcionamento do mecanismo de extensão e da aquisição da gramática de sua *EUPL*.

Como podemos, notar o Modelo Semiótico também é mais preciso e expressivo que o paradigma de programação imitativa, uma vez que ele não somente leva em conta a ordem das ações do usuário, mas também disponibiliza uma *EUPL* que permite ao usuário expressar condicionais, iterações implícitas e explícitas e variáveis. Além disso, ele garante que todos os elementos que pertençam à parte extensível da *UIL* (a *UILx*) tenham uma representação na *EUPL*, seja ela textual ou a própria *UIL*.

É interessante notar que também é possível integrar, em uma única aplicação, mecanismos que realizam programação imitativa com os que realizam o Modelo Semiótico. Em particular, o mecanismo de programação por demonstração (PD), que emprega exemplos das tarefas realizadas pelos usuários na *UIL* para capturar o conjunto de ações que virão a constituir uma extensão, é um mecanismo muito útil em um ambiente de extensão. O uso de exemplos é cognitivamente atraente para os usuários, pois facilita a expressão de sua intenção semântica por meio do emprego de um mecanismo que ele já conhece — a *UIL* da aplicação. No entanto, devido à necessidade de identificar condicionais, iterações e variáveis, um grande número de exemplos é necessário para gerar uma extensão, o que faz com que sua facilidade de uso decaia muito. Assim, é possível visualizar um mecanismo que faça uso de PD para capturar a intenção semântica inicial do usuário através de um exemplo gerado na própria *UIL* da aplicação, expressando-a diretamente em uma *EUPL* que seja semioticamente contínua com a parte extensível desta *UIL* (a *UILx*). Numa segunda fase do processo de extensão, o usuário editaria esta versão para complementar a expressão final de sua intenção com os elementos de interação que não puderam ser capturados automaticamente. Deste modo, dada a existência de continuidade semiótica entre a *UILx* e a *EUPL*, o esforço total de programação realizado pelo usuário final seria reduzido de forma significativa, pois boa parte das instruções teriam sido geradas automaticamente. Além disso, o ambiente poderia ainda tirar proveito da *ADKB* para auxiliar os processos de inferências utilizados

nos mecanismos de PD para a identificação de condicionais, repetições e variáveis, por meio do uso do modelo do domínio e das regras de design empregadas pelo designer na construção do software. Uma possibilidade de integração mais avançada destes dois modelos foi explorada no trabalho de Barbosa [BARBOSA '99], desenvolvido no *SERG*, que define a idéia de Programação Via Interface.

O paradigma de programação descritiva, também discutido no Capítulo 2, amplia ainda mais o poder expressivo da *EUPL* entregue aos usuários finais. Ele normalmente emprega uma *EUPL* textual que tem a possibilidade de disponibilizar para o usuário todo o poder computacional de uma máquina convencional. Estas *EUPLs* em geral são derivadas de linguagens completas de programação, apresentando um alto poder computacional (equivalente a uma máquina de Turing), porém com pouquíssimos mecanismos comunicativos. Infelizmente, como vimos no Capítulo 3, a ausência de mecanismos comunicativos torna a usabilidade destas *EUPLs* muito baixa, ficando seu uso quase impraticável para os usuários finais, um público leigo em conceitos computacionais [MYERS '92A]. Por esta mesma razão as linguagens de programação atuais são igualmente inadequadas para a criação de extensões. Além disso, os mecanismos que realizam este paradigma não apresentam um processo de criação de extensão que auxilie o usuário na realização desta tarefa e os seus ambientes de extensão são, geralmente, desconexos da aplicação, dificultando mais ainda a tarefa de realização de extensões.

Novamente, considerando os aspectos semióticos do paradigma de programação descritiva, nosso modelo teórico nos possibilita prever que o problema mais sério desta ausência de limites às formas de expressão na *EUPL* estará na possibilidade da violação do princípio do Contínuo Semiótico. Isto ocorrerá toda vez que a *EUPL* permitir ao usuário criar elementos de interação que não respeitem as regras de design estipuladas pelo designer do software e/ou quando a *EUPL* não realizar o mapeamento de algum elemento da parte extensível da *UIL*. Nos dois casos o mecanismo estará admitindo a geração de aplicações que criem formas de interação desconhecidas do usuário ou que, até mesmo, causem a destruição da mensagem original do designer. Logo, em ambos os casos, a usabilidade e a aplicabilidade do software decairão, pois os usuários finais perderão a compreensão de seu modelo de usabilidade.

Uma importante diferença entre o Modelo Semiótico e o paradigma de programação descritiva está no fato de o primeiro empregar uma abordagem diferente para o desenvolvimento de software e, portanto, para a criação de extensões ao software. O Modelo Semiótico vê o software como uma mensagem unidirecional e única que vai do designer do software para o

usuário, possibilitando tratar o uso de software como um fenômeno de comunicação. O uso desta abordagem modifica o Ciclo Mínimo de Interação das aplicações, conforme demonstrado no Capítulo 3. Esta modificação torna clara a necessidade de que uma *EUPL* apresente mecanismos que permitam expressar as interações entre o usuário e o software que ocorrerão em uma extensão, o que não ocorre na programação descritiva.

Assim, apesar de os mecanismos que realizam o Modelo Semiótico e a programação descritiva apresentarem uma *EUPL* textual, a linguagem-tipo para *EUP* proposta neste trabalho, para o Modelo Semiótico, apresenta mecanismos para a descrição das formas de ativação de uma extensão por meio da interface, assim como para a expressão de diálogos de aquisição e apresentação de dados e para a expressão do tratamento dos erros que podem ocorrer em uma extensão. Estes mecanismos garantem o Ciclo Mínimo de Interação de uma extensão, possibilitando que uma extensão criada pelo usuário tenha a mesma estrutura comunicativa que as funcionalidades primitivas de uma aplicação que siga o Modelo Semiótico. Tais mecanismos não são encontrados nas linguagens de extensão de qualquer um dos mecanismos que realizam os paradigmas discutidos no Capítulo 2.

Outra importante diferença entre as linguagens empregadas nos mecanismos que implementam estes paradigmas e no Modelo Semiótico foi basear a linguagem-tipo proposta em um subconjunto das sentenças imperativas da linguagem natural. Esta decisão, conseqüência de nossa visão do uso de software como um fenômeno de comunicação e representação, tem sido corroborada por uma série de trabalhos recentes paralelos ao nosso. Por exemplo, em trabalhos na área de especificação de requisitos, Fuchs e Schwitter [FUCHS '96] demonstraram que um subconjunto controlado da linguagem natural, isto é, que apresenta sintaxe e semântica formalizáveis, pode ser usado com sucesso na especificação de sistemas não-triviais. A linguagem por eles proposta, denominada *ACE*, é um subconjunto controlado da língua inglesa que é processável e que tem semântica definida sobre a Teoria de Representação de Discurso (*DRT*)[62], uma variante sintática da lógica de predicados de primeira ordem. Esta linguagem emprega um subconjunto de estruturas lingüísticas que apresenta uma dependência com o domínio muito semelhante ao proposto neste trabalho (resultado de nossa análise dos planos expressos em linguagem natural por pessoas no dia-a-dia), confirmando parcialmente a validade da usabilidade do subconjunto aqui proposto.

---

[62] *Discourse representation theory.*

Já os trabalhos de Bruckman e Edwards [BRUCKMAN '99] mostram que, ao ampliar o auxílio ao uso de linguagem natural por parte dos usuários finais, para fins de programação, é possível reduzir o número de erros de programação resultantes da discrepância entre a forma como as pessoas planejam as coisas no dia-a-dia e a forma como elas têm de programar um computador, confirmando nossa convicção de que o uso de linguagem natural seria mais adequado a usuários finais.

É importante citar os trabalhos de Pane *et al.* [Pane '01] no projeto de Programação Natural, cujos resultados reforçam os encontrados em nossa análise dos tipos de estruturas lingüísticas necessárias a uma *EUPL*, confirmando, por exemplo, a necessidade de uso de iteradores implícitos, de regras de produção, de operações sobre objetos estruturados (conjuntos e seqüências), entre outras estruturas. Pane *et al.* ainda cita o trabalho de Galotti e Ganong [GALLOTI '85], que demonstram ser possível ampliar a precisão das especificações em linguagem natural, garantindo que os usuários entendam os limites de inteligência do receptor das instruções, no presente caso, a máquina. Isto pode ser obtido, por exemplo, por meio do emprego de um mecanismo de explicação, que é central para a abordagem do Modelo Semiótico.

Por último, é interessante citar o trabalho de Rader *et al.* [RADER '98] que, apesar de ser resultado do estudo de uma linguagem visual, nos mostra a necessidade do uso de estruturas lingüísticas (como a presença de estruturas frasais na linguagem visual), de operadores sobre conjuntos, e de dependência com o domínio na linguagem visual para ampliar sua usabilidade e aplicabilidade. Tais resultados apontam na mesma direção das estruturas propostas para nossa linguagem-tipo para *EUP*, confirmando, assim, a qualidade de suas estruturas.

Estes resultados confirmam nossa decisão de usar um subconjunto da linguagem natural como linguagem-tipo para *EUP*, mostrando que o uso de mecanismos comunicativos derivados da linguagem natural, como os propostos neste trabalho, propiciam maior facilidade de aprendizagem e uso de uma *EUPL* por parte dos usuários finais. No entanto, eles não podem ser usados como última palavra na validação das estruturas indicadas para a linguagem-tipo para *EUP* proposta neste trabalho, sendo necessários mais estudos empíricos com aplicações completas implementadas sob este modelo para que possamos avaliar a real usabilidade e aplicabilidade da *EUPL* aqui proposta.

Dentre os mecanismos propostos para esta linguagem-tipo destacam-se os de referenciação a objetos, que são baseados na estrutura dos sintagmas nominais presentes no subconjunto das sentenças imperativas da linguagem natural. Este mecanismo faz uso ainda de figuras de linguagem

como, por exemplo, as anáforas e as elipses que são tratadas diretamente neste trabalho, e as metáforas e metonímias que são tratadas por Barbosa em [BARBOSA'99]. Tais estruturas de linguagem valem-se de elementos pragmáticos para facilitar a expressão de referências a objetos no texto e também para manter a coesão e coerência textuais. Ao mesmo tempo, elas ajudam a reduzir a quantidade de texto necessária para a expressão de uma tarefa, fator significativamente importante para a compreensão de uma extensão pelos usuários finais. É importante observar que, assim como na linguagem *ACE*, a linguagem-tipo para *EUP* é analisável por um compilador *top-down*, não requerendo um tratamento computacional complexo. Além disso, é possível empregar o mesmo tipo de mapeamento semântico que o usado em *ACE*, apesar de não ser este o caso em nosso trabalho. Esta facilidade de processamento é alcançada devido às restrições lingüísticas existentes neste subconjunto da linguagem natural.

Deste modo, diferentemente do paradigma de programação descritiva, as *EUPLs* geradas a partir da linguagem-tipo para *EUP* proposta no Modelo Semiótico tem um alto poder expressivo, e um poder computacional controlado, conforme descrito no Capítulo 5. Seu alto poder expressivo é conseqüência do emprego de estruturas lingüísticas provenientes da linguagem natural e do controle de seu poder computacional resultante de seus compromissos ontológicos e epistemológicos. Ontologicamente estas *EUPLs* restringem a ação do usuário, na geração de novos elementos, à criação de elementos derivados do domínio da aplicação e, também, restringem o conjunto de elementos interativos disponíveis na criação de novos elementos de interface. Epistemologicamente, elas restringem a classe de extensões que podem ser criadas pelos usuários finais, subtraindo dos textos possíveis de serem gerados pela suas gramáticas aqueles que não são válidos por não produzirem efeitos situados na linguagem única de interação da aplicação. Estas restrições são fruto do emprego de agentes inteligentes de software no controle do processo de realização de uma extensão e na definição de novos elementos à linguagem de interação da aplicação. Tais agentes fazem uso das regras de design utilizadas pelo designer do software, e armazenadas na *ADKB*, conforme descrito no Capítulo 5, limitando os tipos de elementos conceituais e interativos que podem ser gerados aos *types* pré-definidos pelo designer, evitando, assim, que o usuário crie elementos que possam violar o princípio do Contínuo Semiótico.

Deste modo, o Modelo Semiótico apresenta um poder computacional próximo ao apresentado pelo paradigma de programação descritiva. É importante observar que a limitação do seu poder computacional é intencional, pois as restrições feitas a ele refletem o fato de que este é um modelo para a geração de extensões ao software e não para programação do software. Esta

diferença é fundamental, pois define os tipos de restrições necessárias e suficientes a um modelo para a tarefa de *EUP*. Assim, podemos dizer que um modelo para a tarefa de *EUP* poderá ter um poder computacional tão grande quanto o necessário, desde que não desrespeite os princípios da Abstração Interpretativa e do Contínuo Semiótico, que definem a relação necessária e suficiente entre a *UIL*, a *EUPL* e a *UEL* que compõem uma aplicação extensível.

Além de fornecer uma linguagem-tipo para *EUP* semioticamente contínua com a *UIL* e a *UEL* da aplicação, o Modelo Semiótico provê um processo de criação de extensões totalmente integrado aos mecanismos de manutenção do Ciclo Mínimo de Interação da extensão. Este fato, aliado a um agente de gerenciamento deste processo — uma agenda —, fornece um ambiente de realização de extensões bastante rico e coeso. Tal ambiente está fortemente acoplado à aplicação, facilitando em muito a tarefa de realização de extensões por parte dos usuários finais.

Entretanto, é importante salientar que o presente trabalho tem a limitação de não se aplicar aos softwares que operam por manipulação direta. São necessários mais estudos para saber se o Modelo Semiótico pode ser estendido para este tipo de aplicação. As maiores dificuldades estão em determinar como expressar as extensões que podem ou não serem realizadas sobre as entidades que operam por manipulação direta. Este conhecimento é essencial para determinarmos como construir agentes de construção de interfaces que garantam a manutenção dos princípios da Abstração Interpretativa e do Contínuo Semiótico.

Também é conveniente ressaltar que o Modelo Semiótico foi desenvolvido para as aplicações monousuário. Logo, não é possível prever sua aplicabilidade ao software multiusuário sem maiores estudos que avaliem o papel da comunicação de grupo sobre a tarefa de criação e uso de extensões, tal trabalho foi desenvolvido dentro do *SERG* por da Cunha [CUNHA '01]. Além disso, é relevante o fato de que este modelo se aplica somente aos sistemas adaptáveis [GIRGENSOHN '92], não sendo, portanto, incorporadas à aplicação extensões que não tenham sido intencionalmente geradas pelo usuário.

É interessante observar que os mecanismos de referência a objetos propostos para a linguagem-tipo para *EUP* do Modelo Semiótico fornecem um alto poder de expressão, permitindo que o Modelo Semiótico suporte não somente extensões permanentes (extensões que são armazenadas com a aplicação), mas também uma ampla gama de extensões transientes (que são usadas somente no contexto da tarefa atual do usuário [BARBOSA '99]). Este resultado sugere que toda aplicação extensível poderia ter um mecanismo lingüístico de aquisição de comandos que

possibilitasse ao usuário a expressão destas extensões transientes. Tal mecanismo poderia ser utilizado, juntamente com o mecanismo de histórico da execução, para, posteriormente, auxiliar o usuário na criação de extensões permanentes.

Como podemos ver, a grande limitação dos paradigmas convencionais para a tarefa de *EUP* não está em sua capacidade de gerar extensões, que pode ser muito grande, mas sim na perspectiva que eles adotam para o desenvolvimento de software e, por conseqüência, da criação de extensões. Com exceção feita a alguns trabalhos desenvolvidos na academia, a maioria dos mecanismos desenvolvidos pela indústria de software baseia-se na visão do software como uma ferramenta [KAMMERSGAARD '88]. Esta visão, apesar de necessária ao desenvolvimento de software, não leva em conta os aspectos comunicativos necessários à tarefa de *EUP*. Esta ausência pode ser sentida mesmo em aplicações que não são extensíveis. No entanto, é nas aplicações extensíveis, onde os usuários leigos em conceitos computacionais tentam adaptar as aplicações às suas necessidades, que ela se faz sentir mais fortemente.

Por outro lado, a visão do software como um meio de comunicação é a base do Modelo Semiótico. Tal visão nos possibilita levar em conta uma série de fenômenos que não são considerados na visão convencional gerando, assim, um modelo mais efetivo para a realização da tarefa de *EUP*. Portanto, o Modelo Semiótico vai ao encontro do critério de usabilidade proposto por Adler e Winograd, fornecendo um conjunto de ferramentas que amplia o conhecimento do usuário tanto sobre as funcionalidades atuais de sua aplicação, quanto sobre as possibilidades de extensão destas funcionalidades, suportando, também, o processo de realização destas extensões.

Cabe aqui uma avaliação dos custos envolvidos em nossa proposta. Em testes preliminares, realizados com modelos reduzidos de domínios para um cliente de *e-mail* e uma agenda, pudemos avaliar que uma total re-engenharia destas aplicações é necessária para incorporar os mecanismos propostos na arquitetura que ampara o Modelo Semiótico. Assim, é certo que o custo de desenvolvimento de uma aplicação que siga o Modelo Semiótico seja superior ao de uma aplicação não-extensível, mas não é possível prever diretamente esta mesma relação entre a criação de uma aplicação empregando o Modelo Semiótico e empregando os modelos atuais de aplicações extensíveis. Isto se deve basicamente a dois motivos. Primeiro, não é razoável avaliar somente o custo de uso de um modelo de software extensível, é necessário que comparemos também os benefícios por ele proporcionados aos usuários finais. Em segundo lugar, é igualmente preciso que descontemos do custo de uso do Modelo Semiótico o custo resultante da ausência de familiaridade

com seu emprego. Somente depois de feitas estas ressalvas é que será possível estimar a verdadeira relação custo/benefício do Modelo Semiótico em comparação aos modelos atuais de desenvolvimento de aplicações extensíveis.

Infelizmente, a ausência de descrições mais precisas na literatura sobre as arquiteturas empregadas nos modelos atuais de software extensível [CYPHER '93A] nos impossibilita comparar esta relação em termos teóricos. Deste modo, como trabalho futuro mais imediato, temos a implementação completa do Modelo Semiótico em uma aplicação real como, por exemplo, um cliente de *e-mail*, uma agenda, um editor de texto, entre outras, com a posterior coleta de dados empíricos sobre seu uso na fase desenvolvimento e seu uso pelos usuários finais. Apenas de posse destes dados, será possível avaliar qualitativa e quantitativamente o real custo de nossas propostas teóricas e sua viabilidade comercial. São também necessários estudos mais profundos para determinar a qualidade da linguagem de representação de conhecimento da *ADKB* de modo que ela apresente um bom auxílio à geração de texto. Deste modo, poderemos usá-la diretamente na *UEL* da aplicação. Também sugerimos a necessidade de estudos cognitivos com uma *EUPL* baseada na linguagem-tipo proposta para a realização de refinamentos em seus mecanismos de comunicação e para a introdução de mais figuras de linguagem empregadas em linguagem natural que facilitem a expressão dos usuários finais.

Outra área de estudos que merece atenção é o desenvolvimento dos agentes de construção de interface para que eles possam melhor amparar a tarefa de *EUP*. Nesta área são necessários trabalhos para identificar a melhor forma de adquirir e armazenar as regras de design que são empregadas por estes agentes. Estas regras fazem parte da lógica da aplicação e são usadas pelo designer de software durante a fase de desenvolvimento do software. A captura da lógica de uma aplicação é uma tarefa árdua e há algum tempo tem levantado muitos questionamentos [MORAN '96]. Ainda nesta área, é necessário estudar a amplitude que pode ser atingida pelos agentes de construção de interface quanto à possibilidade de alteração de interfaces que atuam por manipulação direta.

Além destes trabalhos, que afetam diretamente a viabilidade comercial de nossa proposta, é interessante avaliar também a aplicabilidade de nosso modelo a outras áreas da informática. Em particular, levantamos a possibilidade da aplicação da linguagem-tipo proposta neste trabalho na manutenção, e possivelmente na formação, de bases de conhecimento diretamente por parte de usuários finais. Parece-nos razoável que uma *KRL* instanciada a partir da linguagem-tipo proposta

neste trabalho (acompanhada de um ambiente de aquisição de conhecimento que respeite os princípios da Engenharia Semiótica) apresente características de usabilidade que venham possibilitar a um usuário final dar manutenção a uma base de conhecimento, ou, até mesmo, criá-la.

Outra área promissora para a aplicação da linguagem-tipo desenvolvida neste trabalho é a da engenharia de software. Nesta área, antevemos a possibilidade de aplicar uma variante da linguagem-tipo proposta na fase de análise de requisitos e na aquisição da lógica de design de um projeto de software. Como citamos anteriormente, existem trabalhos que empregam linguagens controladas na fase de análise de requisitos como, por exemplo, a linguagem *ACE* anteriormente citada, que são muito semelhantes em objetivo à linguagem-tipo para *EUP* proposta neste trabalho. Uma modificação necessária à nossa proposta seria o emprego de uma semântica definida por um formalismo semelhante a *DRT* também anteriormente citado. Este tipo de semântica permitiria a transformação direta de uma especificação em programas *Prolog* o que a tornaria uma especificação executável. Esta mesma variante da linguagem-tipo proposta poderia ser ampliada de forma a permitir a documentação das demais fases do projeto possibilitando, assim, a captura da lógica de design de um projeto.

Anexo I

RESULTADOS DOS TESTES COM UMA VERSÃO PRELIMINAR DA LINGUAGEM-TIPO PARA *EUP*

Inicialmente, descreveremos os testes que foram realizados com dois conjuntos de participantes (o primeiro composto exclusivamente por usuários da língua inglesa — totalizando 5 participantes — e o segundo composto exclusivamente de programadores também usuários da língua inglesa — totalizando 5 participantes) com uma versão preliminar da linguagem-tipo para *EUP* proposta neste trabalho e, depois, apresentaremos uma análise dos resultados obtidos que nos levaram a alterar a forma final da linguagem-tipo para *EUP* proposta.

O conjunto de testes realizados seguiu um processo que continha as seguintes etapas:

1. Reposta a um questionário para a caracterização do perfil do participante;
2. Leitura de um texto descritivo do ambiente do Robô Karelzim[63], empregado como domínio para os testes;

---

[63] Karelzim foi baseado no robô Karel empregado por Pattis *et al.* no ensino de programação [PATTIS '94].

3. Execução do Módulo 1 (Avaliação das **formulações naturais** de expressão de planos), em que os participantes deviam descrever, em linguagem natural, os planos para a execução de um conjunto de extensões predefinidas ao domínio do Karelzim;
4. Execução do Módulo 2 (Avaliação da **expressividade** da *EUPL*), em que os participantes deviam descrever, com suas palavras, as tarefas que estavam sendo realizadas por um conjunto de extensões codificadas em *EUPL*[64]; e
5. Execução do Módulo 3 (Avaliação da **produção de textos** em *EUPL*), em que os participantes deviam escrever uma extensão codificada em *EUPL* para um conjunto pré-definido de tarefas.

Os três módulos foram projetados para auxiliar os participantes na aprendizagem da sintaxe e semântica da *EUPL*, visto eles não a conhecerem. O Módulo 1 teve a função de identificar elementos que poderiam estar faltando na *EUPL* e o estilo de programação que melhor se adaptasse a cada tipo de usuário. O Módulo 2 tinha a função primordial de comunicar a sintaxe e a semântica da *EUPL* para os participantes, mas como função secundária, ele serviu para avaliar a facilidade de compreensão do uso da *EUPL*. O Módulo 3 tinha a função de testar o uso da *EUPL* como forma de expressão de extensões realizadas pelos usuários sobre o domínio dado.

A seguir apresentaremos a descrição do domínio e as tarefas e o texto empregados nos três módulos de teste.

1. Descrição do ambiente

A figura abaixo apresenta uma tela descritiva de um estado do KARELZIM, introduzido em Barbosa *et al.* [BARBOSA '00], uma aplicação demonstrativa de como se pode ensinar um pequeno robô imaginário (Karel) a fazer coisas novas em um mundo reduzido (representado pela interseção de 10 avenidas verticais com 10 ruas horizontais), em que estão presentes uns poucos objetos (*beepers* azuis e vermelhos, postes de luz e muros) e configurações especiais resultantes da disposição de tais objetos no mundo, pela ação do robô ou pela definição do usuário.

O robô tem uma capacidade restrita de ação: ele sabe apenas efetuar as ações de **mover-se** um passo à frente, **ir para** uma esquina (x, y), **virar** (à esquerda, à direita, para o norte, sul, leste ou

[64] Várias destas tarefas eram idênticas às apresentadas no Módulo 1. Este padrão foi intencionalmente criado para facilitar a identificação pelo usuário, levando-o ao aprendizado da sintaxe e semântica da *EUPL*.

oeste e para onde estava olhando antes), **colocar** um *beeper* em um local, **colocar** um *beeper* em sua sacola, **contar** coisas (*beepers*, esquinas, etc.) e **verificar** coisas (entre elas: se existe um muro ou poste de luz na próxima esquina e se há *beepers* na mesma esquina em que ele está ou na sua sacola). Mas, Karel pode aprender coisas novas, se elas resultarem de uma combinação especial de coisas que ele já sabe. Por exemplo, ele pode aprender a mover-se rápido, que é, por duas vezes, **mover-se** um passo à frente. Pode também aprender a esvaziar sua sacola, "colocando" na esquina em que está todos os *beepers* que se encontram dentro de sua sacola naquele momento.

Nem sempre uma ação do robô pode ser executada. Não é possível ao robô mover-se para um lugar onde haja um muro ou para fora dos limites do mundo, nem pegar um *beeper* que não exista, nem colocar um *beeper* em algum lugar se ele não dispõe de um em sua sacola. Nestes casos, todas as ações realizadas até o momento pelo robô serão desfeitas, retornando o mesmo ao seu ponto inicial, e o comando produzirá como resultado uma mensagem de erro que será apresentada pela aplicação ao usuário.

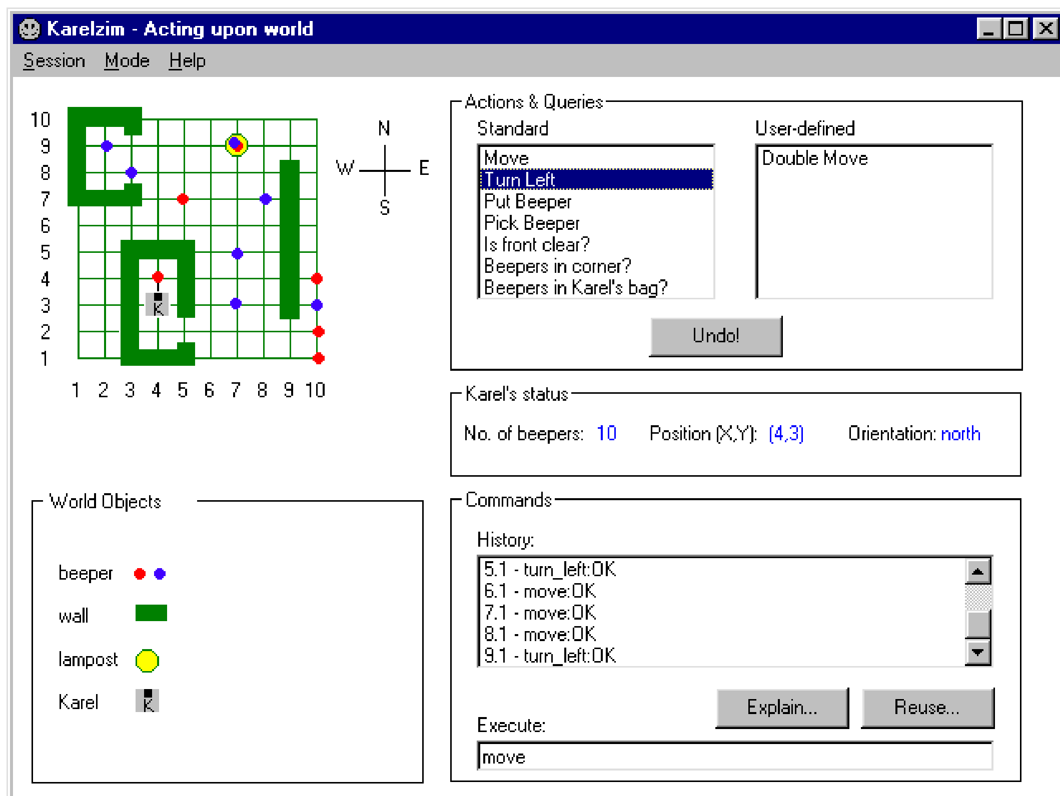

Figura 37: Descrição do ambiente de programação do robô Karezim.

2. Módulo 1. Avaliação das “formulações naturais” de expressão de planos

Nesta parte do teste, dada a situação do mundo descrita na figura anteriormente apresentada, por favor, descreva com suas próprias palavras em português (pelo menos) e em inglês (se possível), qual conjunto de instruções você daria ao robô Karel para que ele realizasse as tarefas abaixo:

1. Contar quantos *beepers* vermelhos existem na avenida 10.
2. Contar quantos *beepers* de certa cor existem em determinado lugar.
3. Contornar o muro que se encontra na avenida 9.
4. Recolher os *beepers* vermelhos que estão ao pé do poste de luz.
5. Trocar a cor de um *beeper* para azul.
6. Colocar todos os *beepers* azuis em uma "caixa" na esquina (2,4). Obs.: Você precisará definir o que é uma caixa no ambiente do Karelzim. Faça isto descrevendo-a por semelhança com outros elementos que já existem no ambiente.

3. Módulo 2. Avaliação da expressividade da *EUPL*[65]

Nesta parte do teste, para cada um dos conjuntos de instruções abaixo, que estão na linguagem de programação do Karel (*EUPL*), por favor, descreva com suas próprias palavras, em português, o que está sendo realizado em cada um deles.

Algumas observações são pertinentes: a expressão fixa **<<*Guess!*>>** está mascarando os nomes de alguns elementos de domínio que dariam a pista da resposta; os demais elementos entre <…> indicam elementos que são variáveis dentro dos comandos; e por último, todo conjunto de instruções, que constitui um novo comando, tem um diálogo de tratamento de erro que deve apresentar uma mensagem esclarecedora ao usuário do erro causado caso o comando não possa ser completado a contento.

Ao final de cada descrição, comente livremente sobre a facilidade de interpretação das instruções da *EUPL*, incluindo sugestões de mudança onde você achar conveniente.

[65] Apresentamos somente três dos treze conjuntos fornecidos aos participantes por motivos de simplificação da descrição.

Conjunto 1)

<<GUESS!>> from <place_1> to <place_2> is a behavior of robot that is activated by the <<GUESS!>> menu or the <<GUESS!>> shortcut, and follows the steps:

1. If <place_1> and <place_2> are not indicated, ask for them using the <<GUESS!>> dialog.

2. If you are not at <place_1>, go there.
3. Pick a "beeper".
4. Go to <place_2>.
5. Put down that "beeper".

6. If there is an error, show the message "<<GUESS!>>", using the error-handling dialog.

Conjunto 2)

<<GUESS!>> is a behavior of robot that is activated by the <<GUESS!>> menu, and follows the procedure:

1. If <place> and <direction> are not indicated, ask for them using the <<Guess!>> dialog.

2. If you are not at "corner (9,2)", go there.
3. Turn east and move.
4. Turn north.
5. Repeat the following three times.
   5.1. Move and turn left.
   5.2. Repeat the following until you are not facing a "wall"
      5.2.1. Turn back and move.
      5.2.2. Turn left.

6. Move.

7. If there is an error, show the message "<<Guess!>>", using the error-handling dialog.

Conjunto 3)

Move <number> times is a behavior of robot that is activated by the move_N_times_ahead menu, and is the result of the following procedure:

1. If <number> is not indicated, ask for it using the move_N_times_ahead dialog.

2. Repeat the following <number> times.
   2.1. Move.

3. If there is an error, show message "It's not possible to move the robot further, there is some barrier in front of it", using the error-handling dialog.

<<GUESS!>> <size> is a behavior of the robot that is the result of the procedure:

1. If <size> is not indicated, ask for it using <<GUESS!>> dialog.

2. Put down a beeper.
3. Move <size> times.
4. Repeat the following three times.
   4.1. Put down a beeper.
   4.2. Turn left and move <size> times.

5. If there is an error, show message "<<GUESS!>>", using the error-handling dialog.

4. Módulo 3. Avaliação da produção de textos em *EUPL*

Nesta parte do teste, por favor, apresente, para cada uma das tarefas abaixo, um conjunto de instruções que oriente o robô Karel a realizá-las. Para tal, você deverá usar uma linguagem tão próxima quanto você possa conseguir da linguagem de programação do Karel (*EUPL*). Você pode consultar os exemplos empregados na seção passada.

1. Colocar três *beepers*, que estão na sua sacola, na rua 2 entre a avenida 7 e a 10.

2. Colocar *beepers* em todas as esquinas de uma rua, que será escolhida por quem comanda o robô. Obs.: Supor que existam *beepers* suficientes na sacola.
3. Colocar todos os *beepers* de um conjunto de cores, indicado por quem comanda o robô, em um certo lugar, que também será indicado por quem comanda o robô.
4. Colocar todos os *beepers* da sua sacola numa caixa na esquina (6,9). Obs.: É necessário criar uma caixa nesta posição.
5. Contar todos os *beepers* vermelhos e colocá-los numa caixa na esquina (3,6) e os azuis ao pé do poste, guardando os demais na sua sacola.
6. Mudar o formato de um *beeper* para quadrado. Obs.: *Beepers* não têm formato até o presente momento, desta forma você deverá definir este novo atributo e um comportamento que permita ao robô mudá-lo.

5. Análise dos testes com a versão preliminar da *EUPL*

Apresentaremos agora um resumo dos resultados obtidos em cada módulo dos testes e uma análise destes resultados, procurando apresentar exemplos tirados diretamente do texto gerado pelos usuários testados.

5.1. Módulo 1

1. Tanto os usuários finais quanto os programadores, a princípio, não se preocuparam com a entrada de dados e saída dos resultados. Além disso, os programadores podem ter presumido a existência de parâmetros para as funções, concentrando-se somente no núcleo do procedimento.

Tarefa 2) Contar quantos beepers de certa cor existem em determinado lugar.

a) Resultado de um usuário final.

```
Go to 3 street and check if there are yellow beepers.
Count the yellow beepers in 3 street.
```

B) Resultado de um programador.

```
FUNC dois (color)
 Counter <- 0
 FOR EACH Beeper in the corner? DO
       IF Beeper in the corner IS color ? THEN
           Add 1 TO counter
```

2. Os usuários finais normalmente fizeram uso do plural para representar a noção de **muitos**. O que indica a necessidade de um tratamento morfológico para as referências.

Tarefa 1) Contar quantos beepers vermelhos existem na avenida 10.

```
Go to Avenue 10.
Count the red beepers in avenue 10.
```

3. Os usuários finais pensaram em função de *tokens*, visto sempre indicarem as ações em função de instâncias, mesmo quando se pede para que seja feito um exemplo genérico (tarefa 2), ou seja, que se faça o uso de *types*.

```
Go to Avenue 5.
Are there blue beepers in Avenue 5?
If it is an affirmative answer, count the blue beepers.
```

4. A forma como os usuários definiram a caixa na tarefa 6 demonstra o uso de **definição por demanda**, que é comumente usada por eles.

```
Pick all the blue beepers.
Go to the corner 2,4.
There you will find a box, something used to keep things, as a bag. Put all the
blue beepers there.
```

5.2. Módulo 2

1. Os usuários finais identificaram com boa precisão as tarefas realizadas no módulo 1.
2. Os usuários finais sugeriram que certas combinações de comandos, como, por exemplo — *turn left and move* —, podem ser convertidas em um único comando, neste caso — *move left*.
3. A maioria dos usuários finais testados tiveram dificuldades com o comando de **atribuição**.
4. A alteração de um atributo de um objeto é sentida como sendo realizada pelo próprio usuário.
5. A noção de repetição muitas vezes pode ser transformada em uma noção de deslocamento dentro de uma estrutura, por exemplo — *move <size> times* → *move <size> corners*.
6. Os usuários finais tiveram dificuldades em interpretar as ações genéricas do exemplo 4 (contornar o muro, que não foi apresentado neste apêndice) e as ações que continham variáveis.
7. Alguns usuários finais questionaram a necessidade de testar se o robô está em um lugar antes de ir para lá. A sugestão foi de que a implementação deveria tratar deste fato, ou emitindo uma mensagem de que ele já se encontra lá ou simplesmente ignorando a ação (pois já foi realizada).

5.3. Módulo 3

1. Os usuários finais não assimilaram bem o conjunto de comandos válidos para o robô. A maior probabilidade para este problema é o pouco tempo de contato com o ambiente e a linguagem, visto que a produção de texto com a linguagem foi satisfatória.

2. A maioria dos usuários finais construiu somente a parte do núcleo da tarefa, não se preocupando com a definição do "nome do comando", nem com os "parâmetros" que devem ser usados.

```
Tarefa 4) Colocar todos os beepers da sua sacola numa caixa na esquina (6,9).
Obs.: É necessário criar uma caixa nesta posição.

Box 1 is a box at corner (6, 9)
---- falta ao cabeçalho do comportamento ----
If <place> <container> are not indicated, ask for them using the <<guess>>
dialog.
Go to box 1
Put all beepers in the box
```

3. Quando um usuário final apresentou alguma preocupação com a necessidade da etapa de definição ficou claro que ele somente a introduziu por obrigação, pois não fez uso correto da nomeação dos comportamentos.

```
Tarefa 2) Colocar beepers em todas as esquinas de uma rua, que será escolhida
por quem comanda o robô. Obs.: Supor que existam beepers suficientes na sacola.

<<guess>> is a behaviour of the robot that is activeted by the <<guess!>> menu,
and follows the steps below:
If <object> and <place> are not indicated, ask for them using the <<guess!>>
dialogue.
If you are not at <place>, go there.
Turn north.
Repeat the following up to the last corner of <place>.
 Put down a beeper at the corner.
 Move.
If there is an error, show the …
```

4. Não houve também, em geral, preocupação com o tratamento de erros e, quando o comando para esta tarefa foi introduzido, ele não se encontrava completo, conforme mostra o exemplo acima.

5. Assim como no módulo 1, a maioria dos usuários e programadores não se preocupou com a entrada e saída dos valores.

```
Tarefa 5) Contar todos os beepers vermelhos e colocá-los numa caixa na esquina
(3,6) e os azuis ao pé do poste, guardando os demais na sua sacola.

If <object>, <color 1>, <color 2>, <container> and <place> are not indicated,
ask for them using the <<guess>> dialog
For all beepers in the world, go to their places, check them count them and
pick them.
Show the counter of red beepers, using the <<guess>> dialog.
Go to box 1
Put all the red beepers in the box
Go to the lamppost
Put down all the blue beepers under the lamppost.
```

6. O fato de a maioria dos usuários que utilizaram a cabeçalho ou inseriram o comando para o tratamento de erros não tê-lo feito de forma completa corrobora com a necessidade de uso de um auxílio na escrita dos comandos longos.
7. O fato de nem todos os comandos terem ficado completos corrobora com a idéia de que é necessário um editor que suporte avaliação tardia[66] de forma a informar ao usuário final sobre a falta de informação na construção dos comandos.

Como principais resultados destes testes podemos tirar as seguintes conclusões:

- Os usuários (sejam eles finais ou programadores) não se preocupam com a etapa de entrada e saída de dados, nem com o tratamento de erros. É importante salientar que este comportamento pode ter resultado do fato de os participantes do teste estarem respondendo a uma **situação de discurso**, onde prevalecem pressuposições e implicaturas importantes para a inteligibilidade mútua. Como a pergunta ou tarefa está **centrada na ação**, a questão de entrada e saída de dados pode ser delegada à pressuposição. As pessoas sabem que deve haver entrada e saída de dados, mas pelas próprias regras de conversação propostas por Grice [GRICE '75], não é preciso falar sobre isto, pois estariam violando a máxima de quantidade[67]. Assim sendo, será papel do ambiente e do processo **topicalizar** (isto é, introduzir como tópico de conversação) os itens que não podem ser pressupostos pelo preposto do designer, forçando o uso correto destes elementos por parte dos usuários.
- É necessário que uma *EUPL* apresente um tratamento morfológico para dar suporte ao uso de plural na representação da noção de muitos e de repetições.
- É necessário que uma *EUPL* tenha um alto grau de articulação para suprir a forma de definição por demanda que é comum entre os usuários finais (o público alvo deste trabalho).
- Como a maioria dos testes indicou uma correta identificação das tarefas no Módulo 2, podemos dizer que a estrutura global da linguagem-tipo para *EUP* proposta tem boa

---

[66] *Lazy evaluation.*

[67] A máxima da quantidade diz que a contribuição do falante não deve informar nem mais nem menos do que o necessário [GRICE '75]

legibilidade e é de fácil compreensão, dois elementos essenciais para obtermos uma boa usabilidade.

- Os usuários finais apresentam grande dificuldade no uso de comandos de atribuição. Este fato sugere que uma *EUPL* deva, de preferência, empregar uma semântica declarativa, que é mais próxima da semântica encontrada no subconjunto da linguagem natural empregada na expressão de planos do dia-a-dia por estes usuários.
- Os usuários (sejam eles finais ou programadores) têm dificuldade no uso de comandos longos. Assim, é necessário o uso de um ambiente com um editor dirigido por sintaxe ou que faça o uso de *templates* de modo a auxiliar os usuários a lembrarem a sintaxe dos comandos longos e também a definição de seus elementos.
- O grão dos comandos primitivos tem grande influência sobre a compreensão das extensões. Um grão muito fino leva os usuários a sentirem-se entediados com o baixo nível da especificação. Um grão muito grosso leva os usuários a não compreenderem a realização da tarefa por não conseguirem ver a cadeia causal entre as ações realizadas.

Anexo II

EXEMPLOS DO USO DE UMA INSTÂNCIA DA LINGUAGEM-TIPO PARA *EUP*

Neste anexo, apresentaremos alguns exemplos de extensões sobre dois domínios (um cliente de *E-mailer* simples e uma Agenda simples), empregando uma instância da linguagem-tipo para EUP proposta neste trabalho. Primeiramente, apresentaremos os modelos dos domínios acima descritos e, depois, as extensões.

1. Exemplo de extensões a um *E-mailer* simples

A Figura 38 apresenta o modelo estático[68] de um *E-mailer* simples com uma única caixa de correio. Este *e-mailer* possibilita que o usuário receba e envie mensagens por endereços diferentes (com ocorre em todos os sistemas deste tipo). Ele permite que as mensagens tenham *attachments* e que o seu *header* seja visível ou não. É possível salvar uma mensagem em uma área específica e também procurar por um texto dentro de uma mensagem específica. É relevante salientar que este é o modelo de usabilidade do *E-mailer*, ou seja, é o modelo do software percebido pelo usuário. Os

[68] Na notação gráfica da linguagem de especificação *UML*.

atributos de cada entidade e as suas ações primitivas também estão representados no modelo. É importante lembrar que o grão de especificação das ações primitivas tem grande influência na compreensão que o usuário desenvolve do sistema, conforme as conclusões dos testes realizados com uma versão preliminar da linguagem-tipo para *EUP* proposta, descritas no Anexo I.

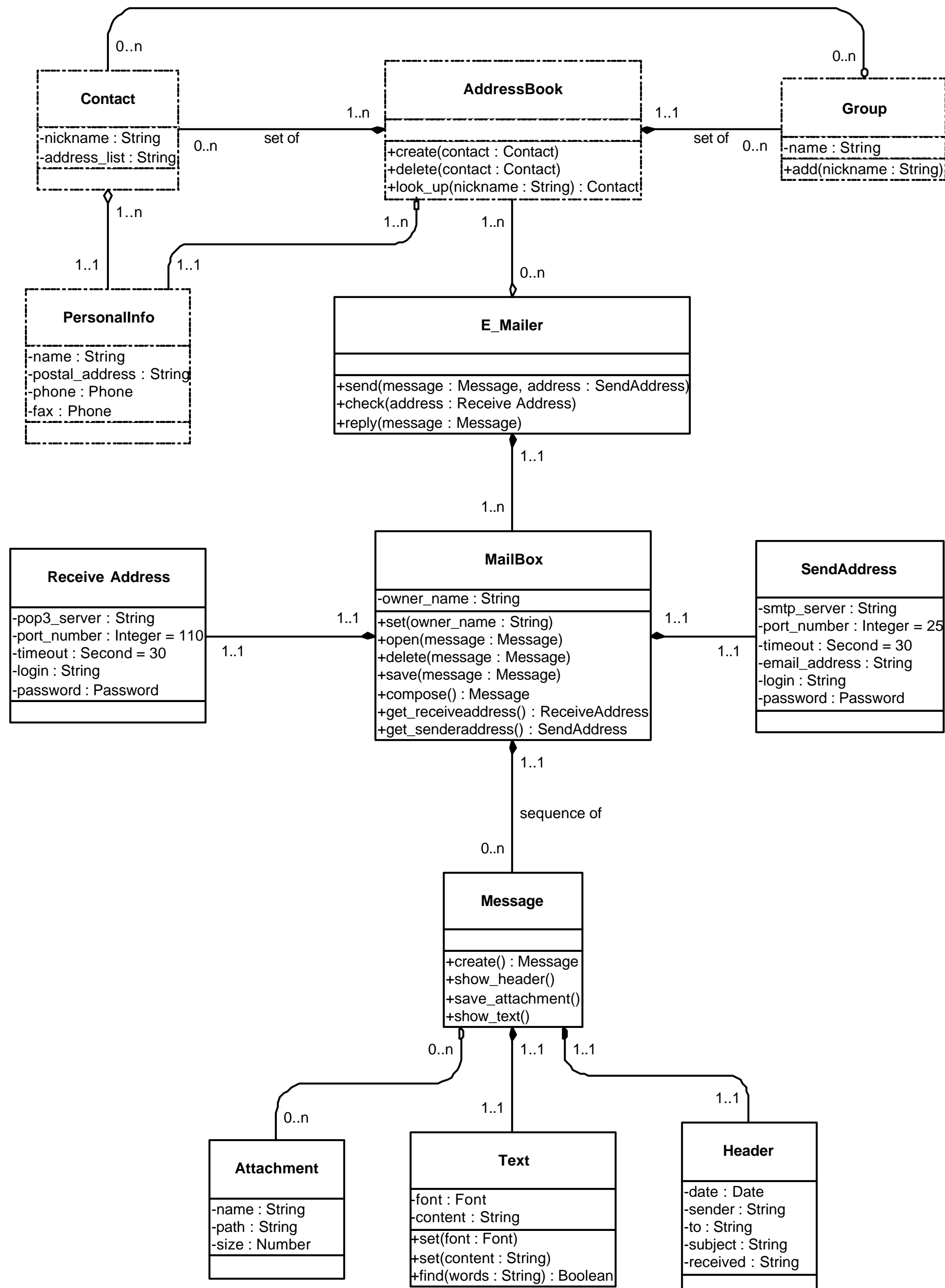

Figura 38: Modelo conceitual de um *E-mailer* simples, conforme visto por um usuário final.

A seguir, apresentaremos um conjunto de extensões possíveis de serem realizadas sobre este domínio com o uso de uma *EUPL* instanciada a partir da linguagem-tipo para *EUP* proposta.

O texto que segue apresenta algumas anotações com os seguintes significados: 1) os elementos marcados com --- podem ser omitidos do texto sem causar prejuízo à sua análise sintática e, portanto, à sua compreensão e 2) os elementos marcados com --- e com fonte em itálico indicam entidades ou ações que são predefinidas na implementação da *EUPL*.

1. **Criação de um caderno de endereços (*AddressBook*) para o *E-mailer*** (a criação desta entidade implica a necessidade de criar as entidades contatos — *Contacts* — e informação pessoal — *PersonalInfo* — para organizar os elementos do modelo conceitual do usuário).

% Extension code for the definition of a PERSONAL INFO
A personal-info **is an** *entity* **that has a** name**,** **a** postal-address**,** **a** phone-number **and a** fax-number**.** **The** name **and the** postal-address **are** *strings*. **The** phone-number **and the** fax-number **are** *numbers*.

% Extension code for the definition of a CONTACT
A contact **is an** *entity*. **It has a** nickname**,** personal-info **and an** address-list**.** **The** nickname **and the** address-list **are** *strings*.

% Extension code for the definition of an ADDRESS BOOK
An address-book **is an** *entity*. **It has** owner-info**, which is** personal-info**.** **It has a** set **of** contacts**.** **It is** part **of an** e-mailer**.**

2. **Criação de grupos (*groups*) para classificação dos contatos dentro do caderno de endereços**.

% Extension code for the definition of a GROUP for the address book
A group **is an** *entity* **that has a** set **of** contacts**.** **It has a** name**, which is a** *string*.

% Extension code to link the GROUP to an ADDRESS BOOK
An address-book **has a** set **of** groups**.**

3. **Criação da ação de cadastramento de um contato no grupo**.

% Extension code for an action to set the <group> of a contact
To set **the** contact's <group> **is an action of the** contact **that is activated by** the *menu* **and follows these instructions:**
If the <group> **belongs to the** <address-book>, add **the** contact **to the** <group>**.**
If not, *show* **the** *message* "The specified group does not exist." **and** *ask for* **the** <creation-of-a-new-group> *using* **the** create-group *dialog*.
If <creation-of-a-new-group> **is accepted,** create **the** <group> **in the** address-book **and** add **the** contact **to the** <group>**.**
If not, *indicate* **an** *error*.
When the <group> **is not** *indicated*, *ask for* **it** ***using*** the get-data *dialog*.
When there is an *error*, *show* the *message* "Some error has occurred while adding the contact to the group. Please, check you data and try again" ***using*** the *error-handling dialog*.

4. Criação da ação de envio de uma mensagem para um grupo.

% Extension code for an action to send a <message> to a <group>
To send a <message> to a <group> is an action of the agenda. It is activated by the menu and follows these instructions:
If the <group> belongs to the <address-book>, send the <message> to all contacts of the <group>.
If not, show the message "The specified <group> does not exist. Please, check your data and try again.".
When the <message> or the <group> are not indicated, ask for them using the get-data dialog.
When there is an error, show the message "Some error has occurred during the sending of the messages. Please, check you data and try again." using the error-handling dialog.

5. Criação da ação de re-encaminhar uma mensagem (*forward* de uma mensagem).

% Extension code for an action to forward the <message>
To forward a <message> is an action of the agenda. It is activated by the menu or by the pop-up menu and follows these instructions:
Ask for the <e-mail-addresses> using the get-address dialog and send the <message> to all the <e-mail-addresses>.
When the <message> is not indicated, ask for it using the get-data dialog.
When there is an error, show the message "Some error has occurred during the sending of the messages. Please, check you data and try again." using the error-handling dialog.

6. Criação da ação de responder a todos os integrantes (reply-to-all).

% Extension code for an action to replay-to-all
To reply-to-all is an action of the agenda. It is activated by the menu or by the pop-up menu and follows these instructions:
Edit the <message> and send it to all the addresses from the receiver of the message.
% Ou
Edit the <message> and send it to all the <message's> receivers.
When there is an error, show the message "Some error has occurred during the sending of the messages. Please, check you data and try again." using the error-handling dialog.

2. Exemplo de extensão a uma Agenda simples

A Figura 39 apresenta a tela principal de uma **Agenda** simples que permite ao usuário anotar compromissos para cada dia da semana em uma única região de texto.

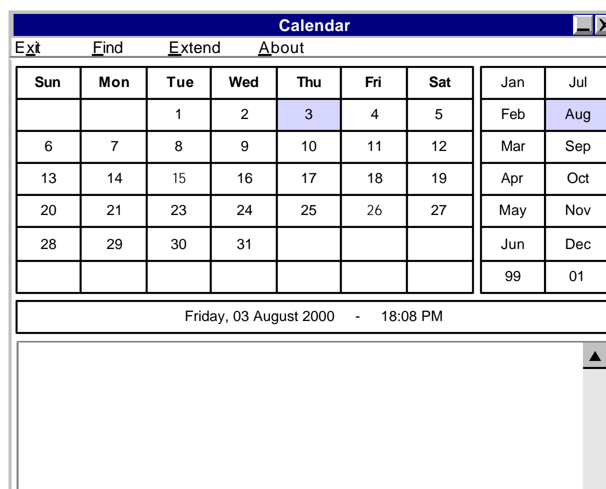

Figura 39: Tela principal de uma Agenda simples com mecanismo de extensão.

Esta agenda também permite ao usuário identificar os dias do mês que apresentam compromissos por meio de sinalização no seu calendário. Este calendário permite sinalizar os feriados de um determinado ano. A agenda contém um mecanismo de extensão marcado no seu menu com o rótulo **Extend**, conforme mostra a Figura 39.

A Figura 40 apresenta um modelo conceitual desta agenda, apresentando as relações entre as entidades visíveis ao usuário na sua interface, assim como seus atributos e ações pré-defindas.

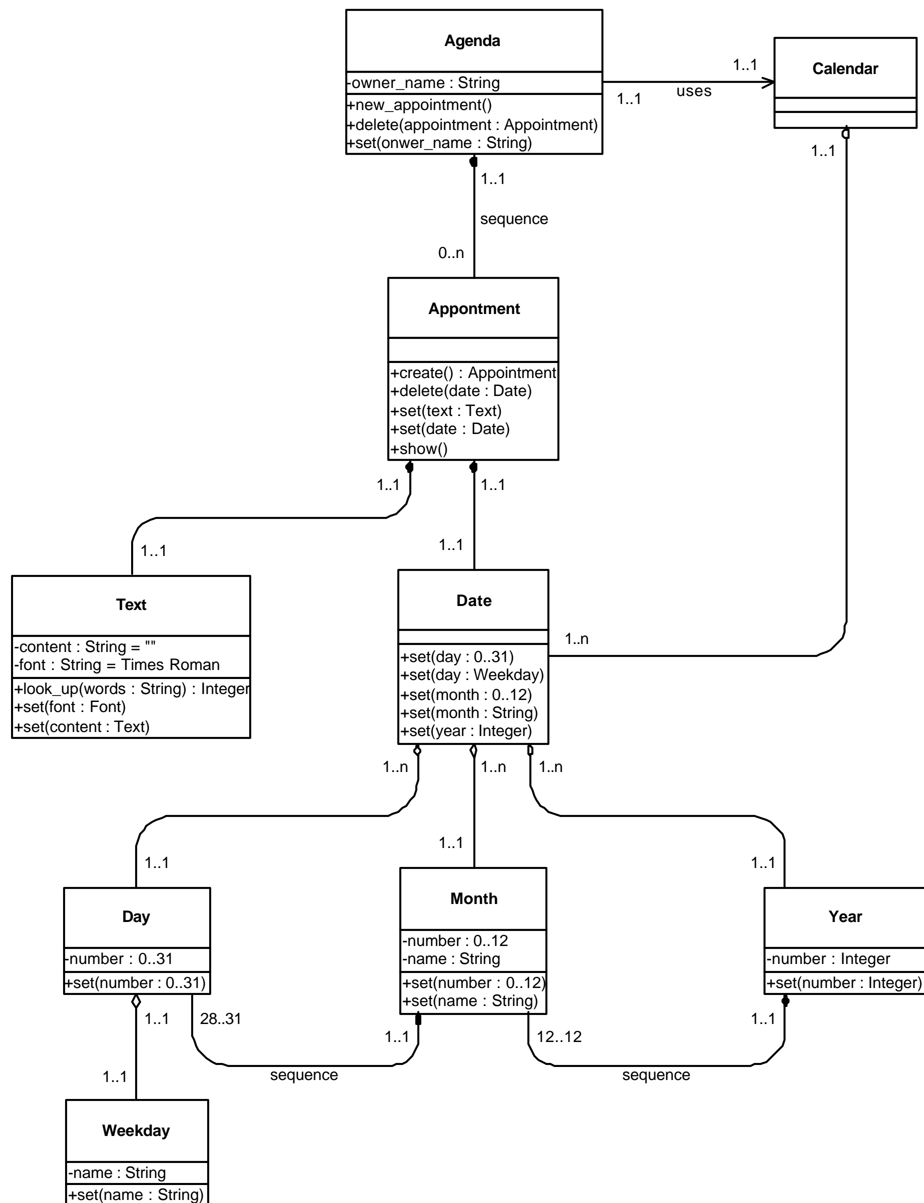

Figura 40: Modelo conceitual de uma Agenda simples, conforme vista por um usuário final.

A seguir, apresentaremos um conjunto de extensões possíveis de serem realizadas sobre este domínio com o uso de uma *EUPL* instanciada a partir da linguagem-tipo para *EUP* proposta.

1. **Criação de uma ação de busca (look up) de um conjunto de palavras em toda a agenda**.

% Extension code for an action to look up <words> in the whole agenda
To look up <words> **is an action of the** agenda **that is activated by the** *menu*. It **follows these instructions:**
For all the appointments **in the** agenda, **if you find the** <word> **in the** appointment, *show* **it.**
If not, *show* the *message* "No appointment was found that contains this text".
When the <words> **are not** *indicated*, *ask for* **them** *using* the get-data *dialog*.
When there is an *error*, *show* the *message* "Some error has occurred during the find process. Please, check your data and try again." *using* **the** *error-handling dialog*.

2. **Criação de uma ação para transferir um compromisso (*appointment*) para outra data**.

% Extension code for an action to transfer an <appointment> to a <new-date>
To transfer **an** <appointment> **to a** <new-date> **is an action of the** agenda **that is activated by the** *menu*. **It follows these instructions:**
Set **the** <appointment's> date **to the** <new-date>.
When the <appointment> **and the** <new-date> **are not** *indicated*, *ask for* **them** *using* the get-data *dialog*.
When there is an *error*, *show* the *message* "Some error has occurred during the transfer process. Please, check your data and try again." *using* **the** *error-handling dialog*.

3. **Adição de tipos aos compromissos**.

% Extension code for adding a type to the appointments
An appointment **has a set of** types **whose default value is "".**

4. **Criação de uma ação para mostrar os compromissos de um tipo**.

% Extension code for show the appointments of a <type>
To show **all** <type> appointments **is an action of the** agenda **that is activated by the** *menu*. **It follows these instructions:**
Get **all the** <type> appointments **of the** agenda **and** show **them.**
When the <type> **is not** *indicated*, *ask for* **it** *using* **the** get-data *dialog*.
When there is an *error*, *show* **the** *message* "Some error has occurred during the selection process. Please, check you data and try again." *using* **the** *error-handling dialog*.

5. **Adição de cores aos dias**.

% Extension code to add a <color> to the days
A day has a color **whose values are** "white", "green", "red", "yellow", "cyan" **and** "magenta". **Its default value is** "white".

6. **Criação de uma ação para definir a cor de um dia**.

% Extension code to set the <color> of a day
To set **the** color **of a** <day> to <color> **is an action of the** agenda **that is activated by the** *pop-up menu*.
It follows these instructions:
Set **the** <day's> **color to** <color>.
When the <day> **and** <color> **are not** *indicated*, *ask for* **them** *using* **the** get-data *dialog*.
When there is an *error*, *show* **the** *message* "Some error has occurred during the setting of color for this day. Please, check you data and try again." *using* **the** *error-handling dialog*.

7. **Criação de uma ação para colorir os compromissos de um determinado tipo**.

%   Extension code to set <type> appointments to <color>
To set **the** <type> appointments to <color> **is an action of the** agenda **that is activated by the** *menu*. **It follows these instructions:**
For all <type> appointments **of the** agenda, *set* **its** day's color **to** <color>.
When the <day> **and** <color> **are not** *indicated*, *ask for* **them** *using* **the** get-data *dialog*.
When there is an *error*, *show* **the** *message* "Some error has occurred during the setting of color for this day. Please, check you data and try again." *using* **the** *error-handling dialog*.

8. **Criação de uma ação para repetir um compromisso todo dia *x* de um ano.**

%   Extension code for repeat an <appointment> every <day> of an year
To repeat **an** <appointment> every <day> **of the** year **is an action of the** agenda **that is activated by the** *menu*. **It follows these instructions:**
For every <day> **of this** year, *create* **the** <appointment>.
When the <appointment> **and** <day> **are not** *indicated*, *ask for* **them** *using* **the** get-data *dialog*.
When there is an *error*, *show* **the** *message* "Some error has occurred during the creation of the appointment. Please, check you data and try again." *using* **the** *error-handling dialog*.

3. Observações gerais sobre o ambiente de extensões

Conforme foi citado anteriormente, existe um conjunto de elementos que devem ser fornecidos pela implementação de um software extensível para que o sistema de extensão funcione de forma satisfatória.

Assim, para cada entidade definida no modelo conceitual do usuário, o sistema deverá fornecer automaticamente comportamentos que permitam:

- **Definir** o valor do atributo ou parte (caso em que deverá ativar a ação de definição da entidade parte) → ***set***;
- **Acessar** o valor do atributo ou parte (caso em que deverá ativar a ação de acesso à parte por completo) → ***get***;
- **Criar** uma nova instância desta entidade → ***create***;
- **Eliminar** uma instância desta entidade → ***remove***;
- **Acessar** uma instância da entidade como um todo → ***get*** e
- **Apresentar** a entidade na interface → ***show***.

Caso a entidade seja um agregado, será necessário que o sistema crie automaticamente ações para realizar o **caminhamento** pela sua estrutura. Assim, são essenciais as ações para:

- **Acessar o primeiro** elemento → ***first***;

- **Acessar o último** elemento → ***last***;
- **Acessar o** elemento **atual** → ***current***,
- **Acessar o** elemento **anterior** → ***previous***;
- **Acessar o próximo** elemento → ***next***;
- **Adicionar** um novo elemento → ***add*** *xxx* ***to*** *yyy*;
- **Retirar** um elemento → ***remove*** xxx ***from*** yyy;
- **Acessar o** agregado **como um todo** → ***get***;
- **Apresentar** um elemento → ***show***; e
- **Apresentar** o agregado **como um todo** → ***show.***
- **Ordenar** o agregado por um atributo → ***sort*** xxx ***by*** yyy.

É necessário também que as entidades: ***Action***, ***Button***, ***Entity***, ***Dialog***, ***Error***, ***Interface***, ***Menu***, ***Message***, ***Shortcut***, ***Pop-up menu***, ***Pull-down menu*** sejam predefinidas. E que as ações ***show*** e ***ask*** também sejam predefinidas.

REFERÊNCIAS BIBLIOGRÁFICAS

[ABELSON '93] ABELSON, H. AND SUSSMAN, G.J. WITH SUSSMAN, J. (1993). *Structure and Interpretation of Computer Programs*-2nd Edition. The MIT Press. Cambridge, MA.

[ADLER '92] ADLER, P. AND WINOGRAD, T. (1992). The Usability Challenge, In ADLER, P. AND WINOGRAD, T. *Usability: Turning Technologies into Tools*. Oxford University Press. New York, NY. p.3-14.

[ALLEN '95] Allen, J. (1995). *Natural Language Understanding* 2nd Edition. The Benjamin/ Cummings. Redwood City, CA.

[ANDERSEN '90] ANDERSEN, P.B. (1990). *A Theory of Computer Semiotics*. Cambridge University Press. Cambridge, UK.

[ANDERSEN '93A] Andersen, P. B. (1993). A Semiotic Approach to Programming. In ANDERSEN, HOLMQVIST AND JENSEN (Eds.) *Computers as Media*. Cambridge University Press. Cambridge, UK.

[ANDERSEN '93B] ANDERSEN, P.B.; HOLMQVIST, B. AND JENSEN, J. (Eds.) (1993). *Computers as Media*. Cambridge. Cambridge University Press, UK.

[APPELT '85] APPELT, D.E. (1985). *Studies in Natural Language Processing: Planning English Sentences*. Cambridge University Press. Cambridge, U.K.

[BARBOSA '97A] BARBOSA, S.D.J.; DE SOUZA, C.S. AND LUCENA, C.J.P. (1997). Supporting Metaphorical Mappings to Facilitate End-User Programming. In Lucena, C.J.P. (Ed.) *Monografias em Ciência da Computação*. Departamento de Informática. PUC-Rio. Rio de Janeiro. MCC/30/97.

[BARBOSA '97B] BARBOSA, S.D.J.; CARA, M.P.; CEREJA,J.R.; CUNHA, C.K.V.; DE SOUZA, C.S. (1997). Interactive Aspects in Switching between User Interface Language and End-User Programming Environment: A Case Study. In *Proceedings of WOMH'97*. São Carlos, Brazil.

[BARBOSA '98] BARBOSA, S.D.J.; DA CUNHA, C.K.V.; DA SILVA, S.R.P. (1998). *Knowledge and Communication Perspectives in Extensible Applications. Proceedings of IHC'98*. Maringá, Brazil.

[BARBOSA '99] BARBOSA, S.D.J. (1999). *Programação via Interface*. Tese de doutorado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil. Maio de 1999.

[BARBOSA'00] BARBOSA, S.D.J.; SILVA, S.R.P. DE SOUZA, C.S. (2000). Extensible software applications as semiotic engineering laboratories, In PERRON, P.; DANESI, M.; UMIKER-SEBEOK, J.; WATANABE, A. (Eds.) *Semiotics and Information Sciences*. Legas Press, Toronto. p.77-96.

[DE BEAUGRANDE'81] DE BEAUGRANDE, R. AND DRESSLER, W. (1981). *Introduction to Text Linguistics*. Longman, London, UK.

[BIBER '99] BIBER, D.; JOHANSSON, S.; LEECH, G.; CONRAD, S. AND FINEGAN, E. (1999). *Longman Grammar of Spoken and Written English*. Longman, Lodon, UK.

[BLOOM '98] BLOOM, P. (1988). Theories of artifact categorization. *Cognition*, № 66, p.87-93.[BRACHMAN '77] BRACHMAN, R.J. (1977). What's in a Concept: Structural Foundations for Semantic Networks. In *International Journal of Man-Machine Studies*. Vol. 2, p.127-152.

[BROWN '83] BROWN, G. AND YULE, G. (1983) *Discourse Analysis*. Cambridge University Press. New York, NY.

[BRUCKMAN '99] BRUCKMAN, A. AND EDWARDS, E. (1999). Should we Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style programming Language. In *Proceedings of the CHI'99*. Pittsburgh, PA. ACM Press. p.207-214.

[CHANDLER '01] CHANDLER, D. (2001): *Semiotics: The Basics*. To be published by Routeledge. Atualmente: *Semiotics for Beginners*. URL: http://www.aber.ac.uk/media/Documents/S4B/.

[CHANG '90] CHANG, S. (1990). *Principles of Visual Programming Systems*. Prentice-Hall. Englewood Cliffs, New Jersey.

[CORDY '92] CORDY, J.R. (1992). Why the Use Interface Is *Not* the Programming Language—and How It *Can* be. In MYERS, B. (Ed.) *Languages for Developing User Interfaces*. Jones and Bartlett, Boston. p.91-100.

[CUNHA'01] CUNHA, C.K.V. (2001) *Um Modelo Semiótico dos Processos de Comunicação Relacionados à Atividade de Extensão à Aplicação por Usuários Finais*. Tese de Doutorado. Departamento de Informática. PUC-Rio. Rio de Janeiro, Brasil.

[CYPHER '93A] CYPHER, A. (1993). *Watch What I Do: Programming by Demonstration*. The MIT Press. Cambridge MA.

[CYPHER '93B] Cypher, A.; Kosbie, D.S. and Maulsby, D. (1993). Characterizing PBD Systems. In CYPHER, A. *Watch What I Do: Programming by Demonstration*. The MIT Press. Cambridge, MA.

[DA SILVA '96] DA SILVA, S.R.P. (1996). *Guidelines for an UIL☯EUPL for word processors.* SERG Technical Report. PUC-Rio. Rio de Janeiro, Brazil.

[DA SILVA '97A] DA SILVA, S.R.P., BARBOSA, S.D.J. AND DE SOUZA, C.S. (1997). Communicating Different Perspectives on Extensible Software. In LUCENA, C.J.P. (Ed.) *Monografias em Ciência da Computação*. Departamento de Informática. PUC-Rio. Rio de Janeiro. MCC 23/97.

[DA SILVA '97B] DA SILVA, S.R.P.; DE SOUZA, C.S.; IERUSALIMSCHY, R. (1997). A Communicative Approach to End-User Programming languages. In LUCENA, C.J.P. (Ed.) *Monografias em Ciência da Computação*. Departamento de Informática. PUC-Rio. Rio de Janeiro. MCC 47/97.

[DE SOUZA '00] de Souza, C.S.; Prates, R.O.; Barbosa, S.D.J; Edmonds, E.A. (2000) Semiotic Approaches to User Interface Design. In *CHI 2000 Extended Abstracts*. The Hague, The Netherlands, 2000

[DE SOUZA '01] DE SOUZA, C.S; BARBOSA, S.D.J AND DA SILVA, S.R.P. (2001). Semiotic Engineering Principles for Evaluating End-user Programming Environments. To appear in *Interacting with Computers*. Vol. 454 (4).

[DE SOUZA '93] DE SOUZA, C.S. (1993). The Semiotic Engineering of User Interface Languages. *International Journal of Man-Machine Studies*. No. 39, p.753-773.

[DE SOUZA '96A] DE SOUZA, C.S. (1996). The Semiotic Engineering of Concreteness and Abstractness: From User Interface Languages to End User Programming Languages. In ANDERSEN, P.; NADIN, M.; NAKE, F. (eds.) *Informatics and Semiotics. Dagstuhl Seminar Report* No. 135. Schloss Dagstuhl. Germany.

[DE SOUZA '96B] DE SOUZA, C.S. AND BARBOSA, S.D.J. (1996). *End-User Programming Environments: The Semiotic Challenges.* In Lucena, C.J.P. (ed.) Monografias em Ciência da Computação. Departamento de Informática. . PUC-Rio Inf MCC 19/96. Rio de Janeiro.

[DE SOUZA '97] DE SOUZA, C.S. (1997). Supporting End-User Programming with Explanatory Discourse. *Proc. of ISAS'97* .

[DE SOUZA '99B] DE SOUZA, C.S. (1999). *Leading users from interaction into programming: The teaching-centered approach of semiotic engineering.* Unpublished manuscript.

[DE SOUZA E SILVA '96] DE SOUZA E SILVA, M.C.P. E KOCH, I.G.V. (1996). *Lingüística aplicada ao português: sintaxe.* 7ª edição. Cortez Editora. São Paulo.

[DERTOUZOS '92] Dertouzos, M.L. (1992). The User Interface is *The* Language. In MYERS, B. (Ed.) *Languages for Developing User Interfaces*. Jones and Bartlett, Boston. p.31-56.

[DIGIANO '95A] DIGIANO, C. (1995). *Introducing Designers to Programming through Self-Disclosing Tools*. Technical Report CU-CS-770-95, Department of Computer Science, University of Colorado at Boulder.

[DIGIANO '95B] DIGIANO, C. AND EISENBERG, M. (1995). Self-disclosing design tools: A gentle introduction to end-user programming. In *Proceedings of DIS '95 Symposium on Designing Interaction Systems*. 189-197. Ann Arbor, Michigan. ACM Press.

[ECO '76] ECO, U. (1976). *Theory of Semiotics*. Indiana University Press. Bloomington.

[ECO '88] ECO, U. (1988). On truth: a fiction. In Eco, Santambrogio and Violi (Eds.) *Meaning and Mental Representations*. Indiana University Press. Bloomington.

[EISENBERG '94] EISENBERG, M. AND FISCHER, G. (1994). Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance. In *Proceedings of CHI'94 (Human Factors in Computing Systems)*. Boston, MA. p.431-437.

[EISENBERG '95] EISENBERG, M. (1995). Programmable Applications. In *SIGCHI Bulletin*, Vol. 27 (2).

[FISCHER '90] FISCHER, G. GIRGENSOHN, A. (1990). End-User Modifiability in Design Environments. In *Proceedings of the CHI'90 Conference: Human Factors in Computing Systems,*. Seattle, WA. p 183-191.

[FETZER '88] FETZER, J.H. (1988). Program Verification: The Very Idea. In *Communications of the ACM*. Vol. 37 (9), p.1048-1063.

[FUCHS '96] FUCHS, N.E. AND SCHWITTER, R. (1996). Attempto Controlled English (ACE). In *Proceedings of CLAW 96 (First International Workshop on Controlled Language Applications)*. Katholieke Univesiteit Leuven, March.

[GALLOTI '85] GALLOTI, K.M. AND GANONG, W.F. (1985). What Non-Programmers Know About Programming: Natural Language Procedure Specification. In *International Journal of Man-Machine Studies*, No. 22, p.1-10.

[GELERNTER '90] GELERNTER, D. AND JAGANNATHAN, S. (1990). *Programming Linguistics*. The MIT Press. Cambridge, MA.

[GIRGENSOHN '92] GIRGENSOHN, A. (1992). *End-User Modifiability in Knowledge Based Design Environments*. Ph.D. Thesis, University of Colorado at Boulder.

[GOODELL '99] GOODELL, H.; MAULSBY, D.; KUHN, S. AND TRAYNOR, C. (1999). Report of the CHI'99 *Workshop on "End-User Programming and Informal Programming "*.

[GREEN '92] GREEN, T.R.G. AND PETRE, M. (1992). When Visual Programs are Harder to Read than Textual Programs. In *Proc. ECCE-6* (*6th European Conference on Cognitive Ergonomics*). CUD, Rome.

[GRICE '75] GRICE, H.P. (1975). Logic and conversation. In COLE, P. AND MORGAN, J. (eds.) *Syntax and Semantics 3: Speech Acts*. Academic Press. New York.

[GUDWIN '97] GUDWIN, R. AND GOMIDE, F. (1997). An Approach to Computational Semiotics. In *Proceedings of ISAS'97*. NIST, Gaithersburg Md. p.467-476

[HOVY '88] HOVY, E.H. (1988). *Generating Natural Language under Pragmatic Constraints*. Lawrence Erlbaum, Hillsdale, NJ.

[HYPERCARD '93] *Apple© HyperCard® Script Language Guide*(1993). Apple Computer, Inc.

[JAKOBSON '60] JAKOBSON, R. (1960). Closing Statements: Linguistics and Poetics. In SEBEOK, T. (ED.) *Linguistics and Communication*. MIT Press, New York, NY.

[JORNA '96] JORNA, R. AND VAN HEUSDEN, B. (1996). Semiotics of the User Interface. In *Semiotica*, Vol. 109 (3/4), p.237-250.

[KAMMERSGAARD '88] KAMMERSGAARD, J. (1988). Four different perspectives on human-computer interaction. In *International Journal of Man-Machine Studies*. No. 28, p.343-362.

[L3D '01] CENTER FOR LIFELONG LEARNING AND DESIGN (2001). *Glossary of Terms*. University of Colorado at Boulder. URL: http://www-l3d.cs.colorado.edu/~l3d/General/glossary.html

[LAKOFF '90] LAKOFF, G. (1990). *Women, Fire, and Dangerous Things*. University of Chicago Press.

[LEITE '97] LEITE, J.C. & DE SOUZA C.S. (1997). A Framework for the Semiotic Engineering of User Interface Languages. In LUCENA, C.J.P. (Ed.) *Monografias em Ciência da Computação*. Departamento de Informática. PUC-Rio. Rio de Janeiro. MCC 10/97.

[LEITE '98] LEITE, J.C. (1998). *Modelos e Formalismos para a Engenharia Semiótica de Interfaces de Usuário.* Tese de Doutorado. Departamento de Informática. PUC-Rio. Rio de Janeiro. October, 1998.

[LEVINSON '83] LEVINSON, S.C. (1983). *Pragmatics*. Cambridge University Press, Cambridge, UK.

[LG '01] *Linguistic Glossary* (2001). International Linguistics Department of the Summer Institute of Linguistics. Dallas, TX. URL: http://www.sil.org/linguistics/glossary/.

[LIEBERMAN '00] LIEBERMAN, H. (2000). Programming by Example. *Communications of the ACM*. Vol. 43 (3), p.72-74.

[LIEBERMAN '98] LIEBERMAN, H. (1998). Integrating User Interface Agents with Conventional Applications. In *Proceedings of IUI 98*. San Francisco, CA.

[LINDEN '91] LINDEN, T.A. (1991). Representing Software Designs as Partially Developed Plans. In LOWRY, M.R. AND MCCARTNEY, R.D. (Eds.) (1991). *Automating Software Design*. AAAI Press, Merlo Park, CA.

[LOTUS '95] LOTUS DEVELOPMENT CORPORATION (1995). *IBM Smart Suite 96*. Software Documentation.

[LYONS '81] Lyons, J. (1981). *Language and Linguistics*. Cambridge University Press. London, UK.

[MARK '99] Mark, D.M.; Smith, B. and Tversky, B. (1999). Ontology and Geographic Objects: An Empirical Study of Cognitive Categorization. In FREKSA, C. AND MARK, D.M. (EDS) *LNCS 1661 : Spatial Information Theory. Cognitive and Computational Foundations of Geographic Information Science*. Spring Verlag. p.283-298.

[MARTINS '98] Martins, I.H. (1998). *Um Instrumento de Analise Semiótica para Linguagens Visuais de Interface*. Tese de Doutorado. Departamento de Informática. PUC-Rio.

[MINSKY '75] Minsky, M. (1975). A Framework for Representing Knowledge. In Winston, P. (Ed.) *The psychology of computer vision*. New Your: McGraw-Hill.

[MOORE '95] Moore, J.D. (1995). *Participating in Explanatory Dialogs: Interpreting and Responding to Questions in Context*. The MIT Press, Cambridge, MA.

[MORAN '96] MORAN, T.P. AND CARROLL, J.M. (Eds.) (1996). *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associate. Hillsdale, NJ.

[MØRCH '97] MØRCH, A. (1997). Three Levels of End-User Tailoring: Customization, Integration, and Extension. In KYNG, M. AND MATHIASSEN, L. (Eds). *Computers and Design in Context*. The MIT Press. Cambridge. MA. 51-76.

[MS VBASIC '95] MICROSOFT *Visual Basic® Language Reference* (1995). Microsoft Corporation.

[MYERS '92A] MYERS, B.A. (1992). *Languages for Developing User Interfaces*. Jones and Bartlett, Boston.

[MYERS '92B] MYERS, B.A. (1992). Introduction. In MYERS, B.A. (ED). *Languages for Developing User Interfaces*. Jones and Bartlett, Boston.

[MYERS '93] MYERS, B.A.; MCDANIEL, R.G. AND KOSBIE, D.S. (1993). Marquise: Creating Complete User Interfaces by Demonstration. In *Proceedings of the INTERCHI'93: Human Factors in Computing Systems*. Amsterdam, Holland. p.293-300.

[MYERS '96] MYERS, B.A.; MODUGNO, F.; MCDANIEL, R.G.; KOSBIE, D.S.; WERTH, A.; MILLER, R.; PANE, J.; LANDAY, J.; GOLDSTEIN, J. AND GOLDBERG, M.A. (1996). The

Demonstrational Interfaces Project at CMU. *AAAI Spring Symposium on Acquisition, Learning and Demonstration: Automating Tasks for Users.* AAAI Press. Stanford, CA, March 25-27. p. 85-91.

[MYERS '00] MYERS, B.A. (2000). Intelligence in Demonstrational Interfaces. In *Communications of the ACM*. Vol. 43(3), p.82-89.

[NADIN '88A] NADIN, M. (1988). Interface Design: A Semiotic Paradigm. In *Semiotica* Vol 69 (3/4).

[NADIN '88B] NADIN, M. (1988). Interface Design and Evaluation — Semiotic Implications, in HARTSON, R. AND HIX, D. (eds.), *Advances in Human–Computer Interaction*, Vol. 2, p.45-100.

[NAKE '94] NAKE, F. (1994). Human-computer interaction: signs and signals interfacing. In *Languages of Design 2*. p.193-205

[NARDI '93] NARDI, B. (1993). *A Small Matter of Programming*. MIT Press, Cambridge, MA.

[NORMAN '86] NORMAN, D.A. (1986). Cognitive Engineering. In NORMAN, D.A. AND DRAPER, S. (eds.) *User-Centered Systems Design*. Lawrence Erlbaum and Associates. Hillsdale, NJ.

[NÖTH'97] NÖTH, W. (1997). Representation in semiotics and in Computer Science. In *Semiotica*, Vol. 115 (3/4), p.203-213.

[DE OLIVEIRA '99] de Oliveira, O.L., Baranauskas, M.C.C. (1999). Communicating entities: a semiotic-based methodology for interface design. In: *HCI International '99 the 8th International Conference on Human-Computer Interaction.* Munich. *Human-computer Interaction: Ergonomics and User Interfaces*. Lawrence Erlbaum Associates. V.1. p.1237-1241. [PANE '01] PANE, J.F.; RATANAMAHATANA, C.A. AND MYERS, B.A. (2001). Studding the Language and Structure in Non-Programmers' Solution to Programming Problems. In *International Journal of Human-Computer Studies*. To appear.

[PAYNE '91] PAYNE, S.J. (1991). Interface Problems and Interface Resources. In CARROLL, J. (Ed.). *Designing Interactions: Psychology at the Human-Computer Interface*. Cambridge University Press. Cambridge, MA.

[PATTIS '94] Pattis, R.E.; Roberts, J. and Stehlik, M. (1994). *Karel The Robot : A Gentle Introduction to The Art of Programming*. John Wiley & Sons.

[PEIRCE '31] PEIRCE, C.S. (1931). *Collected papers*. Cambridge, Ma. Harvard University Press. (excerpted in BUCHLER, J., ed., *Philosophical Writings of Peirce*. New York: Dover, 1955)

[PRATES '00] PRATES, R.O.; DE SOUZA, C.S; BARBOSA, S.D.J. (2000). A Method for Evaluating the Communicability of User Interfaces. In *ACM Interactions*. p.31-38.

[PRATES '98] PRATES, R.O. (1998). On the *Rationale* of Interface Semiotics for Multi-user Applications. In *Proceedings of ISAS'98.*

[PREECE '94] PREECE, J.; ROGERS, Y.; SHARP, H.; BENYON, D.; HOLLAND, S. & CAREY, T. (1994). *Human-Computer Interaction.* Addison-Wesley, Wokingham, UK.

[QUIRK '72] QUIRK, R; GREENBAUM, S.; LEECH, G. AND SVATVIK, J. (1972). *A Grammar of Contemporary English.* Longman, Essex.

[RADER '98] RADER, C.; CHERRY, G.; REPENNING, A. AND LEWIS, C. (1998). Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages. In *Proceeding of 1998 IEEE Symposium of Visual Languages.* 187-194.

[REPENNING '00A] REPENNING, A. (2000). AgentSheets®: an Interactive Simulation Environment with End-User Programmable Agents, In *Interaction 2000.* Tokyo, Japan.

[REPENNING '00B] REPENNING, A. (2000). Programming by Analogous Examples. In *Communications of the ACM*, Vol. 43(3), p.90-97.

[ROSCH '78] ROSCH, E. (1978). Principles of categorization. In ROSCH, E AND LLOYD, B.B. (eds.) *Cognition and Categorization.* Erlbaum, Hillsdale, NJ.

[SCHANK '78] SCHANK, R.C. AND ABELSON, R.P. (1978). *Scripts, Plans, Goal and Understanding: An Inquiry into Human Knowledge Structures.* Lawrence Erlbaum Associates, Hillsdale, NJ.

[SEARLE '79] SEARLE, J.R. (1979). *Expression and Meanings.* Cambridge University Press. New York, NY.

[SMITH '00] SMITH, D.C.; CYPHER, A. AND TESLER, L. (2000). Novice Programming Comes of Age. In *Communications of the ACM.* Vol. 43 (3), p.75-81.

[SMITH '92] SMITH, D.C. AND SUSSER, J. (1992). A Component Architecture for Personal Computer Software. In Myers, B. (Ed.) *Languages for Developing User Interfaces.* Jones and Bartlett. Boston. p.31-56.

[SOWA '91] SOWA, J. (1991). *Principles of Semantic Networks.* Addison-Wesley. Reading, MA.

[SUCHMAN '87] SUCHMAN, L. (1987). *Plans and Situated Actions.* Cambridge University Press. Cambridge, MA.

[TRIGG '87] TRIGG, R.H., MORAN, T.P., & HALASZ, F.G. (1987). Adaptability and Tailorability in NoteCards. In *Proceedings of IFIP INTERACT'87: Human-Computer Interaction* , 723-728

[WILKINS '95] WILKINS, D.E. AND MYERS, K.L. (1995). A Common Knowledge Representation for Plan Generation and Reactive Execution. *Journal of Logic and Computation.*

[WINOGRAD '75] WINOGRAD, T. (1975). Frame Representation and The Declarative/Procedural Controversy. In *Representation and Understanding: Studies in Cognitive Science*. BOBROW, D. AND COLLINS, A. (EDS.). Academic Press, Orlando.

GLOSSÁRIO

Anáfora: uma co-referência de uma expressão com seu antecedente. O antecedente provê a informação necessária para a interpretação da expressão [LG '01].

Aplicabilidade: a condição sob a qual um artefato (uma função, uma ferramenta, etc.) deveria ser usada [L^3D '01].

Caso: modo de marcar a função sintática de um nome, pronome ou adjetivo na estrutura frasal através da formatação da superfície textual [DE BEAUGRANDE '81].

Ciclo Mínimo de Interação: seqüência mínima de ações necessárias para a realização de uma interação completa entre o software e o usuário.

Código: um sistema procedural de convenções relacionadas para correlacionar *representamens* e interpretantes de objetos em um certo domínio. O código provê uma estrutura dentro da qual os signos fazem sentido: eles são dispositivos interpretativos que são usados por comunidades interpretativas [CHANDLER '01].

Coerência textual: diz respeito às formas nas quais os componentes do mundo textual (isto é, a configuração de conceitos e relações que fundamentam a superfície do texto) são mutuamente acessíveis e relevantes [DE BEAUGRANDE '81].

Coesão textual: diz respeito às formas na quais os componentes da superfície do texto, isto é, os signos realizados, estão mutuamente conectados dentro de uma estrutura. Estes componentes dependem um do outro de acordo com formas gramaticais e convenções, de tal forma que a coesão textual estabelece-se sobre a dependência gramatical [DE BEAUGRANDE '81].

Comunicabilidade: é a propriedade de um software que, de maneira eficiente e eficaz, transmite ao usuário sua intenção de design subjacente e seus princípios interativos [PRATES'00].

Co-texto: o contexto local, formado pela a sentença em que uma anáfora ocorre, mais os contextos anteriores, formado pelas sentenças anteriores à sentença em que uma anáfora ocorre que pertencem ao mesmo texto [BROWN '83].

Elipse: a presença de uma descontinuidade na superfície do texto que tem uma função anafórica similar a uma pró-forma (signos vazios de conteúdo próprio, que podem permanecer na superfície do texto no lugar de uma expressão ativadora de conteúdo mais determinada) [DE BEAUGRANDE '81].

Foco do discurso: um termo que se refere à informação, em uma sentença, que é nova, é de alto interesse comunicativo, é realçada por ênfase, tipicamente ocorre mais no final nas sentenças, e complementa a informação pressuposta tipicamente apresentada no início da sentença [QUIRK '85].

Função conativa: (ou apelativa) função da linguagem que evoca uma ação do seu receptor e, normalmente, está relacionada ao vocativo e ao imperativo na linguagem verbal.

Função expressiva: (ou emotiva) função da linguagem que visa à expressão direta da atitude de quem fala em relação àquilo de que está falando.

Função fática: função da linguagem evidenciada por uma troca profusa de fórmulas ritualizadas cujo único propósito é testar o canal de comunicação.

Função metalingüística: função da linguagem que está direcionada para a necessidade do remetente e/ou emissor verificar se eles estão usando o mesmo código.

Função poética: função da linguagem em que o enfoque da mensagem é ela própria, valorizando sua forma de expressão retórica.

Função referencial: (ou denotativa) função da linguagem que enfatiza o referente informando sobre o objeto do discurso.

Implicaturas: qualquer coisa que é inferida a partir de uma expressão, mas que não é uma condição para sua verdade [LG '01].

Interpretante cristalizado do designer: o interpretante que o designer apresentava no momento da implementação do software.

Interpretante: um pensamento, uma sensação, uma ação ou um outro signo que resulta do processo de interpretação de um signo.

Linguagem-tipo: uma descrição para uma classe de linguagem (uma *type language*).

Universais lingüísticos: refere-se a visão de que, enquanto as linguagens variam em sua estrutura superficial, toda as linguagem são baseadas nas mesmas estruturas universais ou leis básicas. Ao contrário do relativismo lingüístico, os universalistas argumentam que podemos dizer qualquer coisa que quisermos dizer em uma língua, e que qualquer coisa que dissermos em uma língua pode sempre ser traduzida em outra [CHANDLER '01].

Meio: entidade ou substância pode aceitar mensagens em uma faixa específica de formas.

Metáfora: em termos semióticos envolve um interpretante atuando como um *representamen* referindo-se a outro interpretante bastante diferente, quando existe alguma similaridade ou correlação entre os dois [CHANDLER '01].

Metonímia: é uma figura de linguagem envolvendo o uso de um interpretante para representar outro que é diretamente relacionado a ele ou quase associado a ele de alguma forma, de forma especial a substituição do efeito pela causa [CHANDLER '01].

Polissemia: qualidade das palavras ou textos que apresentam vários sentidos [CHANDLER '01].

Pressuposições: uma crença básica, relacionada a uma expressão, que deve ser mutuamente conhecida ou presumida pelo emissor e pelo receptor da expressão para que ela seja considerada apropriada ao contexto; geralmente permanecerá como uma hipótese necessária se a expressão for mudada para a forma de uma assertiva, negativa ou questão;

e pode geralmente estar associada a um item lexical específico ou um traço gramatical (ativador da pressuposição) na expressão [LEVINSON '83].

Pragmática: o estudo dos aspectos de significação e uso de linguagem que são dependentes do emissor e do receptor e de outras características do contexto da expressão, tais como o efeito que contexto da expressão, princípios gerais de comunicação observados e os objetivos do emissor têm sobre a escolha da forma de expressão do emissor e a interpretação do receptor de uma expressão [LEVINSON '83].

Representamen: é o "objeto perceptível" (não necessariamente material) que serve como signo para o receptor [PEIRCE '31].

Semiose ilimitada: processo naturalmente ilimitado de geração de uma cadeia de significados.

Semiose: o processo de interpretação de um signo.

Signo: o produto de uma relação triádica entre um objeto, seu *representamen*, e seu interpretante.

Signo frasal: um construto gramatical significativo de uma dada linguagem — isto é, uma estrutura organizadora de signos léxicos.

Signo léxico: uma palavra significativa de uma dada linguagem.

Signo potencial: uma palavra ou frase não-existente em uma linguagem que, no entanto, pode ser gerada por extensões léxicas e/ou gramaticais que obedeçam os seus padrões derivacionais — isto é, as suas meta-regras morfológicas e gramaticais.

Signo realizado: uma palavra ou frase existente em uma linguagem.

Sintagma: unidade sintática de natureza relacional, constituída de uma estrutura binária, cujos membros se relacionam para cumprir a função de comunicação da linguagem [CHANDLER '01].

Texto: um sistema de signos (na forma de palavras, imagens, sons e/ou gestos) . Um texto é construído e interpretado com referência às convenções associadas com um gênero e um meio de comunicação particular, sendo o produto de um processo de representação que "posiciona" ambos seus criadores e seus leitores [CHANDLER '01].

Tópico: uma frase substantiva que expressa o elemento ao qual uma sentença está fazendo referência e ao qual o resto da sentença está relacionada como um comentário [LEVINSON '83].