Este exemplar corresponde à redação final da Tese/Dissertação devidamente corrigida e defendida por: Renato de Oliveira Stehling
e aprovada pela Banca Examinadora.
Campinas, 29 de maio de 2000
COORDENADOR DE PÓS-GRADUAÇÃO
CPG-IC

Projeto e Implementação de uma Arquitetura Reflexiva para a Linguagem Xchart

Renato de Oliveira Stehling

Dissertação de Mestrado

UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

UNICAMP
BIBLIOTECA CENTRAL

Instituto de Computação
Universidade Estadual de Campinas

Projeto e Implementação de uma Arquitetura Reflexiva para a Linguagem Xchart

Renato de Oliveira Stehling

Março de 1999

Banca Examinadora:

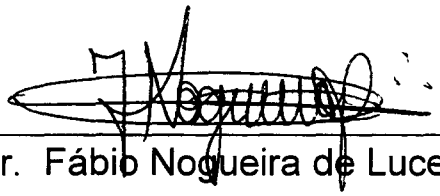
- Prof. Dr. Hans Kurt Edmund Liesenberg (Orientador)
  Instituto de Computação – UNICAMP
- Prof. Dr. Fábio Nogueira de Lucena
  Instituto de Informática – UFG
- Prof. Dr. Luiz Eduardo Buzato
  Instituto de Computação – UNICAMP
- Profa. Dra. Cecília Mary Fischer Rubira (Suplente)
  Instituto de Computação – UNICAMP

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas, como requisito parcial para a obtenção do título de mestre em Ciência da Computação.

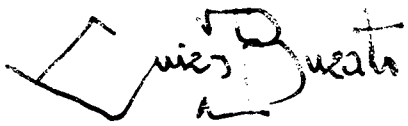
UNICAMP
BIBLIOTECA CENTRAL
SEÇÃO CIRCULANTE

TERMO DE APROVAÇÃO

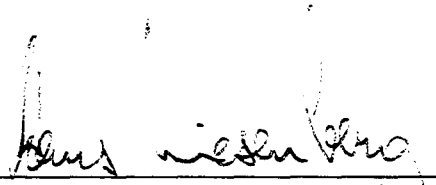
Dissertação defendida e aprovada em 12 de março de 1999, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Fábio Nogueira de Lucena
UFG



Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP



Prof. Dr. Hans Kurt Edmund Liesenbergo
IC - UNICAMP

UNICAMP
BIBLIOTECA CENTRAL

em memória da minha avó Angelina

Agradecimentos

À Deus que permitiu, direcionou, protegeu e me deu forças a cada novo dia.

À minha esposa Eliana e ao meu filho Matheus pelo carinho, compreensão, incentivo, e pelas horas cedidas para que esse trabalho pudesse ser realizado.

À toda minha família, sobretudo meus pais (Célia e Rogério), grandes incentivadores e conselheiros.

Ao meu orientador, Prof. Hans K.E. Liesenberg, por sua prontidão, pela forma como conduziu o trabalho e pela confiança depositada em mim.

À todos os amigos que compartilharam comigo alguns momentos de tristeza e os vários momentos de alegria ao longo dos últimos dois anos.

Aos professores e funcionários do Instituto de Computação pelos serviços prestados e pela disposição em ajudar.

À Fundação de Amparo à Pesquisa do Estado de São Paulo – FAPESP (processo nº 97/11235-4) e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq (processo nº 131884/97-2), pelo apoio financeiro.

Sumário

INTRODUÇÃO **1**

IMPLEMENTAÇÃO ABERTA E REFLEXÃO COMPUTACIONAL **5**

2.1 O MODELO DE IMPLEMENTAÇÃO ABERTA 7

2.2 METAINTERFACES 8

2.2.1 Tipos de metainterfaces *9*
2.2.2 Tema de uma metainterface *10*
2.2.3 Estilo de uma metainterface *11*
2.2.4 Projeto de metainterfaces *12*

2.3 O MODELO REFLEXIVO 12

2.3.1 Arquitetura reflexiva *13*
2.3.2 Classificação *14*
2.3.3 Interpretadores meta-circulares *15*
2.3.4 Protocolo de metaobjetos (MOP) *16*

XCHART **17**

3.1 O AMBIENTE XCHART 18

3.2 ELEMENTOS BÁSICOS 21

3.2.1 Estado *22*
3.2.2 Transição *23*
3.2.3 Evento *24*
3.2.4 Condição *25*
3.2.5 Ação *25*
3.2.6 Regra *26*
3.2.7 Variável de controle e atividade *27*

3.3 EXEMPLO 27

ARQUITETURA REFLEXIVA PARA A LINGUAGEM XCHART 32

4.1 OBJETIVOS 33

4.2 A ARQUITETURA 34

4.2.1 Diagrama reflexivo 35
4.2.2 Adição de regras 37
4.2.3 Adição de transições e estados 38
4.2.4 Elemento de ação reuse 38
4.2.5 Elemento de ação remove 39

4.3 EXEMPLOS 41

4.3.1 Separação de domínios 41
4.3.2 Intervenção no nível base 43

4.4 TORRE REFLEXIVA 44

4.5 IMPLEMENTAÇÃO 46

4.5.1 Editor de Xchart 46
4.5.2 Compilador TeXchart 47

CONCLUSÕES E TRABALHOS FUTUROS 49

5.1 CONCLUSÕES 49

5.1.1 Implementação Aberta e Reflexão Computacional 49
5.1.2 Arquitetura reflexiva para a linguagem Xchart 51

5.2 TRABALHOS FUTUROS 54

5.2.1 Metodologia de desenvolvimento 54
5.2.2 Habilitação/desabilitação em tempo de execução 54
5.2.3 Metainterface introspectiva 55
5.2.4 Sinalização de eventos para diagramas reflexivos 55

REFERÊNCIAS BIBLIOGRÁFICAS 57

Lista de Figuras

Figura 1 - O modelo de caixa preta 6
Figura 2 – Módulo com múltiplas estratégias para a implementação [8] 7
Figura 3 - Metainterface 8
Figura 4 - Etapas do processo de seleção de uma implementação [24] 9
Figura 5 – Possíveis estilos para uma metainterface [8] 10
Figura 6- Estrutura dos sistemas reflexivos 14
Figura 7 - Instantes típicos da execução de uma instância 19
Figura 8 - Modelo lógico da organização de um sistema 20
Figura 9 - Ambiente Xchart 21
Figura 10 - Hierarquia de estados do exemplo da Figura 12 22
Figura 11 - Interface do jogador X em um determinado instante 28
Figura 12 - Gerenciador de diálogo da interface do jogador X 30
Figura 13 – Uma abordagem simples para a mesclagem de diagramas 35
Figura 14 – Exemplos de adição de regras 37
Figura 15 – Exemplo de adição de transição e estado 38
Figura 16 – Exemplos de alteração de regras 39
Figura 17 – Exemplos de remoção de regras, transições e estados 40
Figura 18 – Exemplo da separação de domínios utilizando o nível reflexivo 42
Figura 19 – Exemplo de intervenção no nível base utilizando o nível reflexivo 43
Figura 20 – Exemplo da mesclagem dos diagramas de uma torre reflexiva 45

Resumo

Xchart é uma linguagem visual, orientada a eventos e projetada para especificar controles de diálogo que não podem ser adequadamente descritos com as atuais propostas baseadas em DTEs (Diagramas de Transição de Estados). A arquitetura reflexiva proposta é uma extensão que não visa adaptar Xchart a nenhum novo contexto. O objetivo é controlar a complexidade do código descrito nessa linguagem, promovendo a separação de domínios e permitindo a intervenção na especificação de forma transparente. Nossa proposta procurou preservar ao máximo as características da linguagem Xchart. As novas estruturas são tratadas exclusivamente em tempo de compilação. O sistema de execução de Xchart (SE) não precisou ser alterado porque o código executável é o mesmo da abordagem convencional. Em tempo de execução, não há *overhead* relacionado ao mecanismo de reflexão adotado.

Abstract

Xchart is a visual and event-oriented language designed to specify dialogue controls that can not be adequatelly described by proposals based on STDs (State Transition Diagrams). The proposed reflective architecture does not intend to adapt Xchart to any new context. The main purpose is to control the complexity of the code described in this language by means of promoting domain separation and allowing an interference with specifications in an transparent way. Our proposal tried to preserve the features of Xchart language as much as possible. The new structures are handled in compilation time. There was no need to adapt the Xchart execution system (ES) because the executable code is the same as the one of the conventional approach. At execution time, there is no overhead related with the adopted reflection mechanism.

Capítulo 1

Introdução

O termo Xchart designa simultaneamente uma linguagem e um ambiente de desenvolvimento. A linguagem tem por objetivo descrever o controle de diálogo da interface de um sistema interativo. O controle de diálogo é o componente da interface responsável pela sintaxe da interação com o usuário, pelo *layout* dos objetos visuais e pela comunicação entre apresentação e aplicação. Xchart é uma linguagem visual. A maioria de seus recursos é representada por elementos gráficos, grande parte herdados da linguagem Statechart [10]. Especificações nas linguagens Xchart e Statechart são léxica e sintaticamente semelhantes. Da mesma forma que Statechart estende os DTEs (Diagramas de Transição de Estados), Xchart altera a semântica de alguns recursos herdados de Statechart, além de acrescentar novos recursos. Xchart possui recursos capazes de descrever interfaces que não são adequadamente contempladas ou não podem ser descritas com as propostas atuais (baseadas em DTEs). O ambiente Xchart é composto por ferramentas que auxiliam os usuários da linguagem Xchart a realizar tarefas típicas, por exemplo, edição de especificações, geração de código e simulação.

Embora esse trabalho estenda os recursos de Xchart, as extensões propostas não visam adaptar Xchart a nenhum novo contexto. O objetivo é oferecer recursos que permitam controlar a complexidade das especificações. Assim, a especificação pode ser compreendida mais facilmente, reduzindo os custos de desenvolvimento e manutenção. Para atingir o objetivo de reduzir a complexidade, foram adicionados recursos que permitem que a especificação seja organizada de acordo com os domínios que a compõem. Cada domínio é descrito de forma modular, de forma que possa ser analisado, compreendido e mantido de forma independente. Essa organização facilita a intervenção de especialistas que, na maior parte dos casos, não precisam conhecer todos os domínios que compõem um sistema. Também foram criados mecanismos que permitem a intervenção no

código descrito em Xchart de forma transparente, preservando a robustez da especificação quando elementos são adicionados/removidos em caráter temporário. Essa situação é comum, por exemplo, durante as etapas de depuração, testes e otimização da especificação. Utilizando os novos mecanismos, é possível preservar o código original e as adaptações realizadas. Assim, as adaptações podem ser habilitadas/desabilitadas tantas vezes quantas forem necessárias, sem nenhum tipo de prejuízo.

A técnica utilizada, tanto para promover a separação de domínios quanto para permitir a intervenção na especificação de forma transparente, é conhecida como reflexão computacional (RC). O modelo reflexivo sugere um novo paradigma para o desenvolvimento de sistemas. Nesse novo paradigma, um sistema é decomposto em pelo menos dois níveis: o nível base, que agrupa a implementação do sistema propriamente dita e o nível reflexivo, que agrupa o código reflexivo que estende o nível base. A generalização do conceito de RC a sistemas diferentes de linguagens de programação é conhecida com reflexão implementacional (RI). Utilizando RI, um módulo ou sistema (ao qual denominaremos cliente) que utiliza os serviços oferecidos por outro sistema tem o poder de intervir na implementação do sistema utilizado através de uma metainterface, ortogonal à interface original e com objetivos distintos. Somente detalhes da implementação considerados relevantes podem ser examinados ou alterados através da metainterface. Conforme será mostrado, a intervenção na implementação pode acontecer em diversos níveis, desde a capacidade de consultar a configuração de componentes da implementação até a capacidade de fornecer componentes inteiramente novos.

Os sistemas que permitem aos seus clientes intervirem em sua implementação são conhecidos como sistemas com implementação aberta. Esses sistemas são mais reutilizáveis extensivamente pois, ao invés de esconder a sua implementação conforme o modelo de caixa preta, eles a expõem de forma controlada, permitindo que os clientes para os quais a implementação não é otimizada possam adequá-la às suas necessidades específicas. A necessidade de abrir a implementação é fruto da impossibilidade de oferecer uma única implementação (ou conjunto de implementações) que satisfaça às necessidades de todos os possíveis clientes de um sistema. Mais especificamente, não é possível fornecer uma interface que seja ao mesmo tempo genérica e eficiente.

Nossa motivação para projetar uma arquitetura reflexiva para a linguagem Xchart baseou-se, principalmente, nas vantagens associadas à reflexão computacional e na inexistência desse tipo de arquitetura em linguagens do paradigma de eventos. Conforme será discutido, o paradigma de eventos favorece a criação de uma arquitetura reflexiva comportamental porque a transferência de controle entre os níveis base e reflexivo pode ser realizada de forma natural, através da interceptação de eventos. A implementação da arquitetura implica em alterações em dois componentes do ambiente Xchart: o editor de

Xchart e o compilador TeXchart. Particularmente, o editor de Xchart é de grande importância pois funciona como um complemento à nossa arquitetura. A organização da especificação em níveis possibilita oferecer aos usuários diferentes "visões" da especificação e avançados recursos de navegação. Assim, qualquer tipo de teste para a validação de nossa proposta requer a existência de um editor de Xchart que explore todo o potencial de nossa arquitetura. O sistema de execução de Xchart (SE) não precisa ser alterado porque o código executável é o mesmo da abordagem convencional (sem arquitetura reflexiva). Dessa forma, em tempo de execução, não há *overhead* associado ao mecanismo de reflexão adotado.

O restante desse texto está organizado como se segue. O capítulo 2 discute detalhadamente os modelos de implementação aberta e reflexivo, estabelecendo a relação entre os dois modelos. É fornecida uma definição precisa para alguns conceitos importantes para a compreensão da arquitetura que será discutida. São discutidos os principais problemas associados ao modelo de caixa-preta e que motivaram a criação dos modelos de implementação aberta e reflexivo. Em relação ao modelo de implementação aberta, as metainterfaces serão discutidas em detalhes (tipos, temas, estilos e alguns princípios de projeto). Em relação ao modelo reflexivo, serão discutidos os conceitos de reflexão computacional, reflexão implementacional, auto-representação, arquitetura reflexiva, interpretadores meta-circulares e protocolos de metaobjetos (MOPs). A principal contribuição desse capítulo é agrupar de forma resumida, porém precisa, todas essas informações.

O ambiente e a linguagem Xchart são descritos, de forma resumida e informal, no capítulo 3. São descritos o editor de Xchart, o compilador TeXchart, o sistema de execução de Xchart (SE), a interface de programação de Xchart (IPX), o modelo lógico da organização de um sistema (instâncias, clientes, estímulos externos e portas) e os elementos básicos da linguagem Xchart (diagramas Xchart, estados, transições, regras, eventos, ações, variáveis e atividades). O capítulo é encerrado com o exemplo de uma especificação. Essa especificação também é utilizada no capítulo 4 para exemplificar a utilização de nossa arquitetura reflexiva.

O capítulo 4 descreve nossa proposta. São apresentados nossa motivação, os objetivos da arquitetura, as operações possíveis utilizando o nível reflexivo (adição e exclusão de estados, regras, transições, e elementos de ação), os processos de compilação e execução da especificação e a noção de torre reflexiva. Também é discutida a implementação da arquitetura proposta.

O capítulo 5 apresenta nossas conclusões e trabalhos futuros. As conclusões são relativas às duas principais contribuições de nosso trabalho: a revisão bibliográfica sobre

reflexão computacional e a arquitetura reflexiva que permite organizar a especificação em domínios e, a partir dessa organização, intervir de forma transparente na especificação. Em relação aos trabalhos futuros, as principais extensões identificadas são as seguintes: (1) o desenvolvimento de uma metodologia que auxilie os usuários na organização da especificação utilizando a arquitetura reflexiva proposta, (2) a extensão do sistema de execução de Xchart (SE) para permitir a habilitação/desabilitação do nível reflexivo em tempo de execução, (3) a sinalização de eventos diretamente para os diagramas do nível reflexivo e (4) a criação de uma metainterface instrospectiva que permita aos clientes que interagem com o gerenciador de diálogo consultar a estrutura e o status da execução da especificação.

Capítulo 2

Implementação Aberta e Reflexão Computacional

O grande desafio para a maioria das tecnologias é o tratamento da complexidade. Nos sistemas de *software*, o conceito de modularidade tem sido amplamente utilizado com esse objetivo. Quando um sistema pode ser dividido em módulos com baixo acoplamento e alto grau de coesão conceitual, sua complexidade pode ser controlada mais facilmente. Tradicionalmente, um módulo pode ser visto como uma caixa-preta, que expõe sua funcionalidade através de uma interface bem-definida e esconde detalhes de sua implementação (Figura 1).

Um módulo deve ser projetado para que possa ser reutilizado em diferentes sistemas. Dessa forma, os custos de desenvolvimento e manutenção podem ser reduzidos. Para que um módulo possa ser reutilizado extensivamente, sua interface deve ser genérica o suficiente para que possa oferecer as funcionalidades requeridas por diferentes clientes. Adicionalmente, a estratégia escolhida para a implementação do módulo deve ser eficiente. A dificuldade de desenvolver módulos reutilizáveis é que os dois requisitos acima são contraditórios: apesar de eficazes, estratégias de implementação genéricas não são necessariamente eficientes em casos específicos. Por outro lado, estratégias de implementação eficientes não são necessariamente genéricas. Ao escolher a estratégia para a implementação de um módulo, o desenvolvedor se compromete com um subconjunto de perfis de clientes {C'}, para os quais a estratégia escolhida é otimizada. O conjunto {C'} é um subconjunto do conjunto {C}, formado por todos os possíveis perfis de clientes. O conjunto {C} é potencialmente infinito. Algumas observações importantes podem ser feitas em relação aos clientes com perfil não contemplado ({C} - {C'}), ou seja, para os quais a estratégia escolhida não é otimizada:

- Apesar do código escrito por esses clientes expressar de forma simples e elegante o comportamento desejado, a performance obtida é insatisfatória.

- Por não poder reutilizar o módulo de forma apropriada, esses clientes são forçados a codificar sua própria versão do módulo com a estratégia mais adequada a seu perfil. Assim, o código do cliente se torna maior, mais complexo e menos robusto, aumentando os custos de desenvolvimento e manutenção do sistema como um todo.

Muitas vezes, para oferecer o máximo de eficiência a um subconjunto de clientes, o projetista otimiza não apenas a implementação do módulo, mas também sua interface. Nesses casos, são ocultadas funcionalidades e informações de estado que não são relevantes para o subconjunto de clientes contemplado. De fato, quanto mais funcionalidades e informações acerca do estado interno de um módulo estiverem disponíveis através de sua interface, mais complexa e ineficiente se torna sua implementação. Essas novas otimizações diminuem ainda mais a reusabilidade dos módulos já que, além da performance insatisfatória, clientes com perfil não contemplado também não terão acesso a eventuais funcionalidades e informações de estado de que possam necessitar.

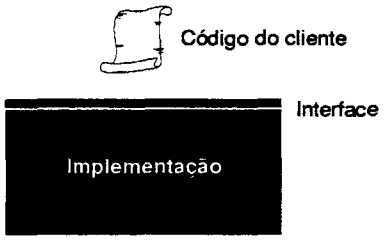


Figura 1 - O modelo de caixa preta

Como exemplo dos problemas associados ao modelo de caixa-preta, considere um módulo que implementa funcionalidades de conjuntos (criação e exclusão de conjuntos, inserção, exclusão e consulta de elementos desses conjuntos) [8]. Existem inúmeras estratégias para a implementação de conjuntos que podem ser aplicadas a esse módulo, por exemplo, *B-trees*, *hash-tables* e listas-ligadas. Determinar a melhor estratégia requer conhecimentos específicos sobre cada conjunto, por exemplo, qual o número mínimo, máximo, e médio de elementos? Qual a freqüência de cada operação (inserção, consulta e exclusão)? Essas informações não podem ser determinadas durante o projeto do módulo. A interface do módulo também não permite identificar essas informações em tempo de execução. Nesse caso, os clientes têm mais condições de determinar a estratégia ideal para a implementação do módulo que o próprio implementador. Utilizando o modelo de caixa-

preta, não há como projetar esse módulo de forma que seja possível reutilizá-lo extensivamente.

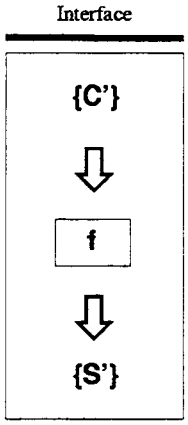


Figura 2 – Módulo com múltiplas estratégias para a implementação [8]

2.1 O modelo de implementação aberta

A prática tem mostrado que um módulo se torna mais útil e reutilizável (extensivamente) se, ao invés de esconder a sua implementação conforme o modelo de caixa-preta, ele a expõe de forma controlada, permitindo a seus clientes ajustá-la às suas necessidades específicas. Esse novo modelo de desenvolvimento é conhecido como **implementação aberta** [25]. O princípio básico do modelo de implementação aberta é que um módulo de *software* pode ser melhor reutilizado se for desenvolvido de forma a acomodar uma ampla faixa de estratégias para a sua implementação. A estratégia mais adequada é selecionada de acordo com o perfil de cada cliente. Como sugere a Figura 2, módulos com implementação aberta são compostos internamente por um subconjunto de perfis de clientes **{C'}**, um subconjunto de estratégias para a implementação **{S'}**, e uma função de mapeamento **f: {C'}→{S'}** que, a cada perfil de cliente, associa a estratégia mais adequada.

2.2 Metainterfaces

O modelo da Figura 2 é o mais básico para módulos com implementação aberta e pressupõe que, ou uma única estratégia é adequada para todos os perfis de clientes, ou o perfil do cliente pode ser determinado automaticamente, de forma que a estratégia mais apropriada a esse perfil possa ser selecionada pela função de mapeamento. Algumas questões relativas a esse modelo são: (1) o que fazer quando o perfil do cliente não puder ser determinado automaticamente? (2) O que fazer quando o perfil do cliente não pertencer ao domínio da função de mapeamento ou, em outras palavras, o que fazer quando não existe uma estratégia apropriada para o perfil do cliente?

Com o objetivo de solucionar as questões identificadas acima, foi incorporado ao modelo de implementação aberta a noção de **metainterface**. Uma metainterface é uma segunda interface adicionada ao modelo de caixa-preta, independente da interface original e com objetivos distintos (Figura 3). A interface original expõe as funcionalidades do módulo, determina como os clientes irão utilizar essas funcionalidades e esconde detalhes da implementação. A metainterface, ao contrário, expõe a implementação do módulo de forma controlada, permitindo aos clientes ter acesso e alterar algumas de suas características. A metainterface não oferece acesso irrestrito à implementação, pelo contrário, a maior parte da implementação permanece oculta, assim como no modelo de caixa preta. Somente detalhes considerados relevantes podem ser examinados e alterados.

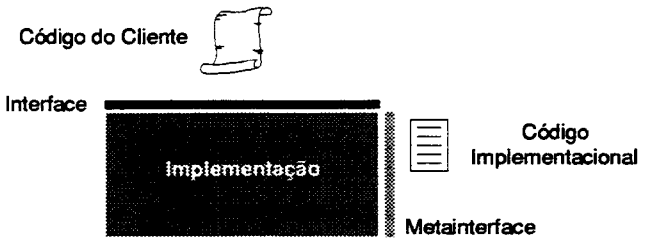


Figura 3 - Metainterface

Os clientes para os quais existem estratégias de implementação adequadas utilizam apenas a interface do módulo, conforme o modelo de caixa-preta. Os clientes para os quais não existe estratégia adequada, além de utilizarem a interface do módulo, também utilizam a metainterface para otimizar a implementação ao seu perfil. Esses clientes acabam escrevendo dois tipos de código: aquele que utiliza da funcionalidade do módulo e que é

relativo ao domínio do sistema que está sendo implementado e um novo tipo de código, ao qual denominaremos **código implementacional**, que otimiza a implementação do módulo ao seu perfil. O código implementacional pode ser declarativo ou programável. O **código declarativo** é simples, mas seu poder está limitado às formas de declaração para as quais a metainterface oferece suporte. O **código programável** supera essa limitação permitindo ao cliente escrever seu código na forma de pequenos programas.

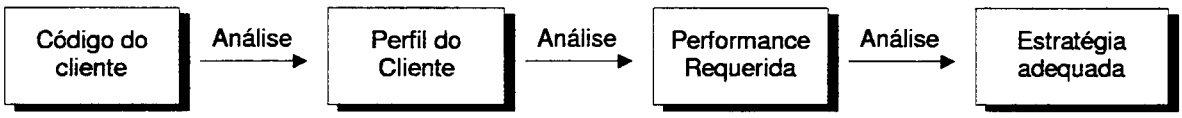


Figura 4 - Etapas do processo de seleção de uma implementação [24]

2.2.1 Tipos de metainterfaces

As metainterfaces podem ser classificadas de acordo com o tipo de acesso à implementação que oferecem [21]. O tipo de acesso mais básico é o de **introspecção.** As metainterfaces introspectivas são utilizadas para permitir aos clientes de um módulo ter acesso a informações de estado que são ocultadas por sua interface. Um exemplo de metainterface introspectiva é o pacote reflexivo incorporado à linguagem Java (JDK1.1) [1]. Quando a interface oculta, além do estado interno do módulo, funcionalidades úteis aos clientes, devem ser utilizadas metainterfaces com capacidade de **invocação**, ou seja, metainterfaces que permitam invocar essas funcionalidades diretamente, contornando as restrições impostas pela interface do módulo. Imagine, por exemplo, que um usuário de C++ necessita criar objetos cujas classes são conhecidas apenas em tempo de execução. Nesse caso, a aplicação da cláusula *new* não é adequada pois requer que o nome da classe seja fornecido em tempo de compilação. Apesar dessa restrição imposta pela interface de C++, certamente existem mecanismos internos que, efetivamente, criam os objetos em tempo de execução e que poderiam ser invocados através de uma metainterface. As metainterfaces mais elaboradas são aquelas com capacidade de **intervenção**. Essas metainterfaces permitem aos clientes intervir na implementação do módulo, adaptando-a às suas necessidades específicas. No restante desse texto, nos concentraremos apenas em metainterfaces com capacidade de intervenção, às quais denominaremos simplesmente metainterface.

2.2.2 Tema de uma metainterface

As metainterfaces têm **temas** associados a si. O tema de uma metainterface diz respeito às características da implementação do módulo às quais essa metainterface se refere explicitamente. Três exemplos de temas são: (1) o perfil do cliente; (2) a performance requerida pelo cliente; (3) a estratégia para a implementação. As metainterfaces cujo tema é o perfil do cliente permitem, por exemplo, que o cliente forneça seu perfil explicitamente. Quando o tema de uma metainterface é a performance requerida, o cliente é capaz de determinar a performance que deseja obter e o módulo se encarrega de escolher a estratégia capaz de oferecer essa performance. No caso de metainterfaces cujo tema é a estratégia para a implementação do módulo o cliente pode, por exemplo, escolher diretamente a estratégia mais adequada para o módulo (dentre as disponíveis) ou então, fornecer uma nova estratégia. A escolha do tema apropriado é uma decisão importante pois afeta a facilidade de desenvolvimento, especificação, implementação e utilização das metainterfaces.

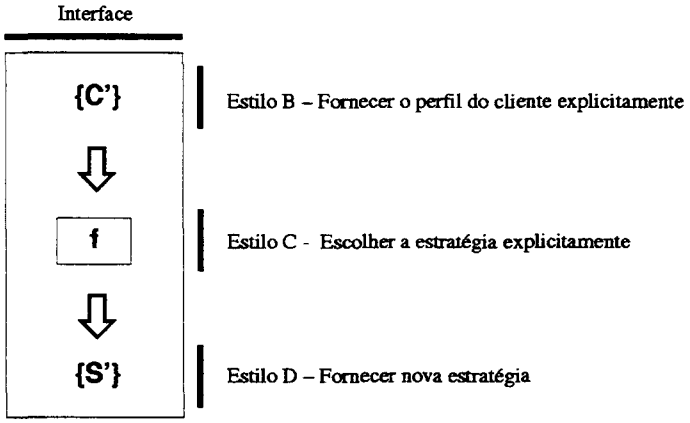


Figura 5 – Possíveis estilos para uma metainterface [8]

Observando-se o processo automático de seleção da estratégia para a implementação de um módulo com implementação aberta (Figura 4), pode-se extrair uma regra que permite escolher o tema apropriado para sua metainterface [24]. Baseado na Figura 4, o melhor tema para a metainterface de um módulo é representando pelo primeiro retângulo, ao longo do processo, no qual valem as seguintes propriedades:

- é possível desenvolver um mecanismo automático que complete o processo a partir do ponto escolhido;

- é relativamente fácil desenvolver uma metainterface que expresse esse tema;
- é fácil para um cliente utilizar essa metainterface.

Vale observar que essa regra também mostra quando as metainterfaces não são necessárias. Uma metainterface não é necessária quando todas as etapas do processo apresentado puderem ser realizadas automaticamente.

2.2.3 Estilo de uma metainterface

As metainterfaces com o mesmo tema podem ter diferentes **estilos**. A Figura 5 mostra três estilos que permitem a um cliente intervir na implementação do módulo de maneiras diferentes [24]. Na prática, a mesma metainterface pode combinar os estilos (e até mesmo temas) que forem necessários através da técnica de *layering*. A separação apresentada é conceitual e tem por objetivo enfatizar que o cliente não necessariamente precisa utilizar, ou até mesmo conhecer, todos os estilos disponíveis em uma metainterface.

O estilo ***A*** é a ausência de metainterface, por isso não aparece na figura. Quando não existem metainterfaces, os módulos selecionam a estratégia mais adequada ao cliente automaticamente. Essa seleção é baseada na forma como os clientes utilizam a interface do módulo. O estilo ***B*** permite aos clientes fornecer seu perfil explicitamente (tema 1). Esse estilo deve ser utilizado quando não é possível determinar automaticamente o perfil do cliente. Observe que as informações fornecidas pelo cliente não determinam a estratégia adotada. A escolha da implementação continua sendo realizado pela função de mapeamento. Em geral, o código implementacional que utiliza metainterfaces do estilo ***B*** é declarativo. O estilo ***C*** permite ao cliente escolher, dentre as estratégias disponíveis para a implementação do módulo, a mais adequada ao seu perfil (tema 3), contornando a função de mapeamento. Esse estilo é útil quando o cliente não é capaz de determinar seu próprio perfil mas conhece a estratégia mais adequada às suas necessidades. O estilo ***D*** permite aos clientes com perfil não contemplado, ou seja, os clientes para os quais não existe uma estratégia otimizada, fornecer uma nova estratégia para a implementação do módulo ou, então, otimizar sua estratégia *default* (tema 3). Um procedimento simples para identificar o estilo mais adequado para uma metainterface (dentre os estilos descritos acima) é:

> ***Se determinar o conjunto {S'} é difícil, utilize o estilo D***
> ***senão, se determinar a função f é difícil, utilize o estilo C***
> ***senão, se determinar o conjunto {C'} é difícil, utilize o estilo B***
> ***senão, utilize o estilo A.***

2.2.4 Projeto de metainterfaces

Após escolhidos os temas e estilos mais apropriados, a metainterface é projetada utilizando-se, por exemplo, a técnica de refinamentos sucessivos [8]. O projeto de metainterfaces deve atingir dois objetivos muitas vezes conflitantes. Por um lado, as facilidades associadas à metainterface não podem prejudicar a implementação efetiva e eficiente da interface do módulo. Por outro lado, a metainterface deve oferecer aos clientes acesso a aspectos da implementação que possam ser utilizados para criar variações semânticas ou otimizações relevantes. A seguir são apresentados alguns princípios que devem guiar o projeto de metainterfaces.

P1. Devem ser expostas apenas informações essenciais sobre a implementação do módulo.

P2. Deve haver uma separação clara entre o código que utiliza a interface do módulo e o código implementacional.

P3. O código implementacional deve ser opcional e fácil de desabilitar.

P4. O escopo de influência do código implementacional deve ser controlado de forma natural e suficientemente granular.

Esses princípios de desenvolvimento têm por objetivos: (1) preservar o máximo possível as virtudes do modelo de caixa-preta; (2) permitir que o código implementacional e o código do cliente que utiliza a funcionalidade do módulo (através de sua interface) sejam compreendidos de forma independente; (3) preservar a integridade do módulo, limitando a propagação de eventuais erros no código implementacional.

2.3 O modelo reflexivo

Um módulo/sistema com implementação aberta permite aos seus clientes utilizá-lo conforme o modelo da caixa-preta quando sua implementação for adequada ou, então, alterar sua implementação quando necessário (nesse caso, o modelo de caixa-preta é enfraquecido). Essa capacidade adicional dos clientes de intervir na implementação do módulo/sistema é conhecida como **reflexão implementacional** (RI). RI é uma visão alternativa (e mais genérica) do conceito de reflexão computacional proposto por Maes [40]. Segundo Maes, **reflexão computacional** (RC) é a atividade realizada por sistemas reflexivos quando esses agem (computam) sobre si mesmos, alterando seu próprio comportamento, estrutura ou status. Sistemas reflexivos são sistemas computacionais que incorporam dados (denominados **auto-representação)** que representam aspectos estruturais e/ou comportamentais do próprio sistema. Um sistema computacional é representado por

um programa e consiste em um processo que é resultado da execução desse programa. RC é a atividade que inspeciona e/ou manipula o processo resultante da execução do programa que representa o sistema. Diversas atividades computacionais são inerentemente reflexivas, por exemplo, qualquer tipo de estatística sobre a execução de um sistema ou facilidades de monitoração e depuração de programas oferecidas pelos atuais ambientes de desenvolvimento. Duas observações sugerem que RC e RI são conceitos equivalentes [43]: (1) o interpretador de uma linguagem, que gera os processos computacionais relativos aos programas, é a implementação dessa linguagem; (2) a interface de qualquer sistema pode ser encarada como uma linguagem cujo interpretador é a implementação do sistema. A primeira observação sugere que RC é um caso especial de RI. A segunda sugere o inverso.

2.3.1 Arquitetura reflexiva

O modelo reflexivo sugere um novo paradigma para o desenvolvimento de sistemas. Nesse novo paradigma, um sistema é decomposto em pelo menos dois níveis, conforme pode ser visto na Figura 6. O **nível base** agrupa as funções relacionadas exclusivamente ao domínio do sistema. O **nível reflexivo** agrupa o código que supervisiona, adapta e retorna informações sobre o nível base (código implementacional ou reflexivo). A decomposição de um sistema em níveis, de acordo com o escopo da computação realizada, apresenta as seguintes vantagens:

- Os sistemas tendem a ser mais estruturados, diminuindo sua complexidade, facilitando sua compreensão e sua manutenção.
- Os sistemas podem ser adaptados de forma transparente.
- Algoritmos de políticas (como distribuição, persistência e concorrência), bem como quaisquer outros algoritmos que não estejam diretamente relacionados ao domínio do sistema, podem ser implementados no nível reflexivo, sem a necessidade de alterações no código do nível base.

De acordo com a abrangência do nível reflexivo, é possível identificar sistemas que oferecem facilidades reflexivas e sistemas que apresentam arquiteturas completamente reflexivas [41]. Os sistemas que oferecem **facilidades reflexivas** permitem aos seus clientes manipular alguns aspectos de sua implementação utilizando um conjunto predefinido de operações. Os sistemas com **arquitetura reflexiva** permitem acesso mais amplo (potencialmente ilimitado) à implementação. Arquitetura reflexiva e implementação aberta são termos utilizados, em contextos diferentes, para caracterizar a organização interna de um sistema. Assim como uma arquitetura reflexiva é pré-requisito para realizar computações reflexivas, uma implementação aberta é pré-requisito para realizar reflexão

implementacional. Dessa forma, arquitetura reflexiva e implementação aberta também são conceitos equivalentes.

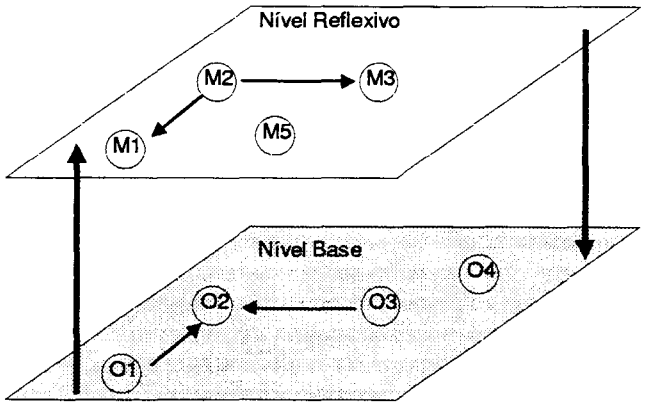


Figura 6- Estrutura dos sistemas reflexivos

2.3.2 Classificação

De acordo com o tipo de auto-representação do sistema, as arquiteturas reflexivas podem ser classificadas em estruturais ou comportamentais [29]. Uma auto-representação estrutural reifica aspectos estruturais de um programa como, por exemplo, tipos de dados e hierarquia de classes. **Reificação** é o processo de converter programas em dados. O processo inverso (converter dados em programas) é denominado **reflexão**[1]. A identificação dinâmica de tipos em C++ (RTTI) [6] é um exemplo de facilidade reflexiva que utiliza uma auto-representação estrutural. Uma auto-representação comportamental reifica o processo computacional e o seu comportamento.

O nível reflexivo de um sistema pode existir em tempo de execução ou em tempo de compilação. As arquiteturas que trabalham em tempo de execução introduzem *overhead* adicional durante a execução de um sistema porque, para executar uma operação no nível

[1] O termo reflexão é utilizado com dois significados distintos nesse texto. Em oposição à reificação, reflexão é a ação de voltar para trás, de virar, modificar a direção. No sentido mais geral (reflexão computacional ou implementacional), reflexão significa ato introspectivo de meditação, ponderação, observação; ato de voltar-se para si mesmo e examinar seu próprio conteúdo.

base, deve ser executado o código reflexivo que a implementa. Um forma de diminuir esse *overhead* é utilizar a técnica de avaliação parcial [36] para diminuir a quantidade de código reflexivo executado para cada operação do nível base. As arquiteturas que trabalham em tempo de compilação não introduzem *overhead* adicional porque não existe código reflexivo em tempo de execução. Nesse tipo de arquitetura, o código reflexivo modifica o compilador para que o programa do usuário seja compilado de uma maneira diferente. O código reflexivo não descreve como um objeto deve se comportar, mas como o compilador deve gerar um código que se comporte de acordo com as intenções do usuário. Enquanto em tempo de execução, a implementação de uma operação do nível base pode ser escolhida de acordo com o contexto de execução, em tempo de compilação, uma única implementação para a operação do nível base deve ser fixada e essa implementação deve satisfazer a todos os contextos de execução previstos.

2.3.3 Interpretadores meta-circulares

Uma característica importante de um sistema reflexivo é que deve haver uma relação causal entre esse sistema e a sua auto-representação, ou seja, alterações no sistema devem implicar em alterações em sua auto-representação, e vice-versa. Grande parte das pesquisas em RC se utilizam de interpretadores metacirculares para satisfazer a essa relação causal. Um **interpretador metacircular** é um interpretador escrito na linguagem que interpreta e que utiliza algumas formas especiais, adicionadas a essa linguagem, que permitem aos programas em execução acessar e modificar o estado de seu interpretador [47]. Os interpretadores metacirculares formam uma torre potencialmente infinita onde o programa do usuário se encontra no nível mais baixo dessa torre. O programa do usuário (nível 0) é interpretado pelo programa do nível 1 que, por sua vez, é interpretado pelo nível 2 e assim sucessivamente. Genericamente, o programa no nível n é encarado como uma estrutura de dados (auto-representação) que pode ser examinada e manipulada no nível n+1. Nesse caso, a relação causal é garantida por definição.

A condição necessária para a utilização de interpretadores metacirculares é que deve haver um formato único para dados e programas, de forma que os programas possam ser encarados como dados e vice-versa. O problema associado a essa abordagem é que a auto-representação deve servir a dois propósitos distintos e com requisitos contraditórios: (1) implementar o sistema (auto-representação efetiva e eficiente) e (2) servir como dado para computações reflexivas (auto-representação expressiva). A seguir, são enumeradas linguagens de programação de diferentes paradigmas, cujas arquiteturas são reflexivas, e que se utilizam de interpretadores metacirculares:

- Linguagens funcionais - 3-LISP [4] e BROWN [9];

- Linguagens baseadas em lógica - FOL [44] e META-PROLOG [34];
- Linguagens baseadas em regras - TEIRESIAS [42] e SOAR [31];

2.3.4 Protocolo de metaobjetos (MOP)

Uma alternativa recente aos interpretadores metacirculares são os MOPs (protocolos de metaobjetos). Um MOP é uma metainterface orientada a objetos. Os objetos da metainterface que correspondem à auto-representação do sistema são denominados **metaobjetos**. Diz-se que os metaobjetos são a reificação de componentes da implementação (que podem ser objetos ou não). Existem diferentes formas de relacionar os objetos do nível base com os metaobjetos do nível reflexivo. Algumas arquiteturas são ditas completamente reflexivas pois existem metaobjetos associados a todos os objetos do nível base (relação 1:1) [40] [49]. Outras arquiteturas associam metaobjetos somente aos objetos que sofrerão algum tipo de computação reflexiva [45], sendo por isso denominadas parcialmente reflexivas. Existem ainda arquiteturas que permitem associar mais de um metaobjeto ao mesmo objeto, em uma relação 1:N [5] [27] [28] [32] [50].

Alguns exemplos de linguagens de programação orientadas a objetos que utilizam MOPs são: SMALLTALK-80 [2], 3-KRS [40], CLOS [22], OpenC++ [45] [46], MetaJava [30] [35], ObjVlisp [39], ABCL/R [49] e KSL [37]. Os MOPs também são utilizados em sistemas diferentes de linguagens de programação. Nesses sistemas, o objetivo dos MOPs é permitir aos clientes intervir de forma controlada em sua implementação (reflexão implementacional). Apertos [51] e Silica [43] são exemplos, respectivamente, de sistema operacional e interface gráfica com implementação aberta e que utilizam MOPs para satisfazer à relação causal entre o sistema e a sua auto-representação.

envolvem diálogos *multithread* (concorrentes), possivelmente distribuídos e empregados no desenvolvimento de sistemas cooperativos é apoiado pelo emprego de Xchart. Em particular, Xchart possui recursos capazes de descrever interfaces que não são adequadamente contempladas ou não podem ser descritas com as atuais propostas baseadas em DTEs (Diagramas de Transição de Estados).

3.1 O ambiente Xchart

Xchart é uma linguagem visual. A maioria de seus recursos é representada por elementos gráficos, grande parte herdados de Statechart [10]. Especificações nas linguagens Xchart e Statechart são léxica e sintaticamente semelhantes. Da mesma forma que Statechart estende os DTEs, Xchart altera a semântica de alguns recursos herdados de Statechart, além de acrescentar novos recursos. A semântica formal de Xchart é descrita por Lucena em [18]. A utilização de Xchart pressupõe o isolamento lógico do controle de um sistema e implica na divisão de um sistema interativo em pelo menos dois subsistemas: o subsistema reativo, responsável pela execução do controle descrito em Xchart, e o subsistema responsável pelas demais funcionalidades (aplicação e apresentação) que estão além dos propósitos de Xchart.

Em Xchart, o controle de diálogo de uma interface é descrito por um conjunto de diagramas denominados **diagramas Xchart**, que servem para a criação de instâncias. Um diagrama Xchart é a menor unidade utilizada para representar controle. A partir de cada diagrama, pode ser criado um número arbitrário de instâncias. A cada instância é associado um identificador único que permite diferenciá-la das demais. As instâncias dos diagramas Xchart interagem entre si (subsistema reativo) e com os demais subsistemas. Em ambos os casos, as informações recebidas por uma instância são denominadas **estímulos externos**. Em um determinado instante, uma instância pode estar ativa, inativa ou inoperante. Quando ativa, uma instância é sensível a estímulos externos, que são acumulados e tratados a medida que ocorrem. Uma instância inoperante não percebe alterações no ambiente externo, não executa ações (não produz controle). Todos os estímulos externos sinalizados para uma instância inoperante são descartados, exceto aquele que contém o evento esperado para ativá-la novamente. Uma instância torna-se inativa quando é finalizada. Instâncias inativas não produzem nenhum tipo de saída.

Quando a vazão de estímulos externos fornecidos a uma instância é maior que a sua capacidade de tratá-los, esses estímulos são acumulados em uma fila de espera. Cada instância possui a sua própria fila de estímulos. Os estímulos são tratados de acordo com a configuração de cada instância. A configuração de uma instância refere-se ao conjunto de estados ativos, valores de variáveis, regras e transições habilitadas e outras informações que

caracterizam a instância em um determinado instante de tempo. O tratamento de um estímulo externo é iniciado com a identificação do conjunto de regras/transições cujos gatilhos estão habilitados. O subconjunto dessas regras/transições que pode ser executado forma um **passo**. Os instantes mais importantes na execução de uma instância são mostrados no diagrama de tempo da Figura 7. Um passo é dividido em um conjunto finito de **micropassos**. A execução do subconjunto de regras/transições cujos gatilhos estão habilitados e que podem ser executadas corresponde ao primeiro micropasso. A execução desse micropasso pode gerar ou não eventos. Quando não são gerados eventos, a execução do passo (composto apenas do primeiro micropasso) é encerrada. Quando a execução do micropasso gera eventos, esses eventos são adicionados ao estímulo externo que está sendo tratado. O estímulo externo acumulado (união do estímulo externo com os eventos gerados no micropasso corrente) realimenta a instância, possivelmente causando a execução de um novo micropasso. Esse processo se repete enquanto a execução do micropasso corrente gera eventos. Essa reação em cadeia é finita porque as regras/transições, assim como a ativação/desativação de estados são executadas apenas uma vez por passo.

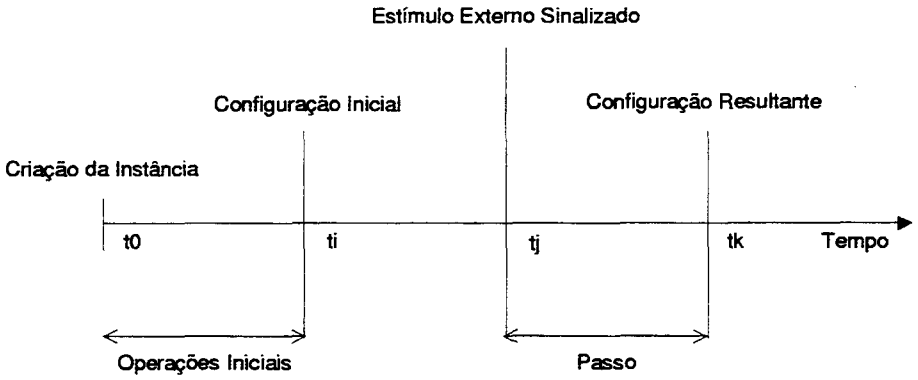


Figura 7 - Instantes típicos da execução de uma instância

As conseqüências de um passo são visíveis externamente após o término de sua execução. As configurações intermediárias da instância (micropassos) não são percebidas pelo ambiente externo, a não ser a configuração resultante da execução do último micropasso (que corresponde ao final da execução do passo). A execução de uma passo pode ter efeitos locais ou externos. Os efeitos locais de um passo terminam quando a execução do passo termina. Os efeitos externos (alteração de variáveis globais, sinalização de eventos paras outras instâncias) somente são percebidos após o término da execução do

passo. O controle produzido por uma instância ao final da execução de um passo é depositado em um repositório denominado **porta**.

Os componentes de um sistema interativo que interagem diretamente com as instâncias dos diagramas Xchart (subsistema reativo) são denominados **clientes**. Esses clientes podem estar distribuídos sobre os nós de uma rede e podem ser executados concorrentemente. Os clientes sinalizam estímulos externos para as instâncias que, por sua vez, depositam suas reações em portas. A partir desse instante, qualquer cliente pode consultar a reação produzida na porta pertinente. A relação entre instâncias, clientes e portas é mostrada na Figura 8. A comunicação entre os clientes e as instâncias de Xchart é realizada através da **IPX** (Interface de Programação de Xchart) [12]. A IPX é uma interface orientada a objetos, utilizada para fazer acesso aos recursos oferecidos pelo Sistema de Execução de Xchart (SE). O desenvolvimento de um cliente envolve a compilação do código do cliente e sua ligação à biblioteca IPX.

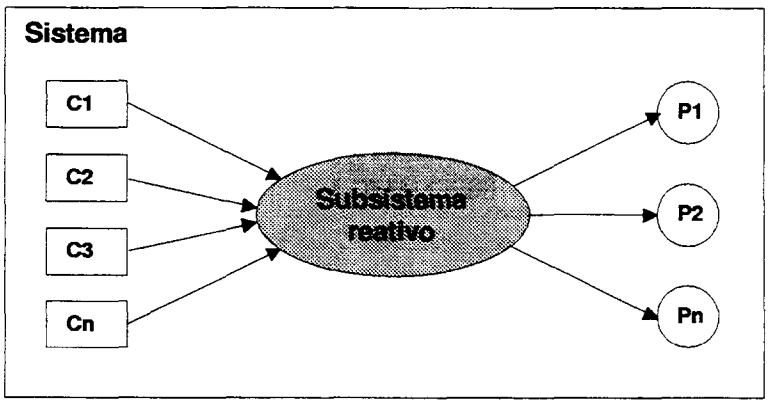


Figura 8 - Modelo lógico da organização de um sistema

Os diagramas Xchart são codificados utilizando-se o Editor de Xchart [38]. O formato visual da descrição na linguagem Xchart é convertido pelo editor, de forma transparente e automática, em um formato texto conhecido como TeXchart [14]. Esse formato é utilizado para armazenar a especificação, para comunicação entre algumas ferramentas do ambiente e também como entrada para o compilador TeXchart. A conversão Xchart/TeXchart é possível nos dois sentidos, sem perda de informação. O compilador TeXchart gera estruturas de dados que serão utilizadas pelo Sistema de Execução de Xchart (SE). Esses dados são armazenados no formato FSDX (Formato de Sistema Descrito em Xchart) [19]. A organização do ambiente Xchart é mostrada na Figura 9. O Sistema de Execução de

Xchart (SE) é composto pelo Gerente de Distribuição (GD) e o Servidor Xchart (SX). O GD é responsável pelo suporte à distribuição (controle de concorrência, localização de recursos, atomicidade). Os serviços fornecidos pelo GD ocultam a possível distribuição de clientes e são utilizados, exclusivamente, pelo SX. As funções do SX são realizadas por dois componentes: O Núcleo Reativo (NR) e o Coordenador. Esses componentes são executados concorrentemente em *threads* (processos leves) distintos. O coordenador mantém dados pertinentes a instâncias em execução, manipula o relógio do sistema, além de outras tarefas dependentes do ambiente operacional. O núcleo reativo (NR) é o componente que executa a semântica formal de Xchart. Dado um estímulo externo e a configuração de uma instância, o NR identifica uma reação e deposita o resultado dessa reação em portas. Eventualmente, o NR utiliza os serviços oferecidos pelo Gerente de Distribuição (por exemplo, requisita *broadcast* de um evento). O NR é o único componente do SE que altera a configuração de uma instância.

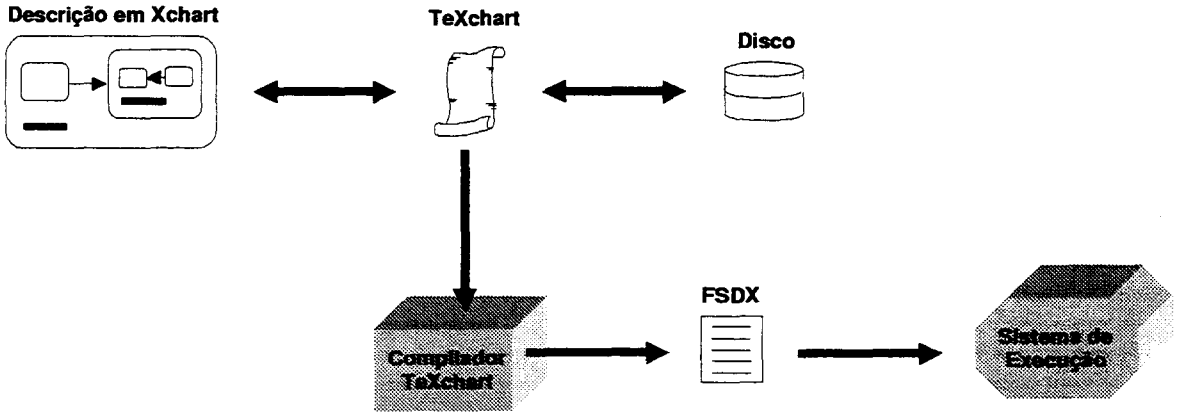


Figura 9 - Ambiente Xchart

3.2 Elementos básicos

Essa seção descreve, informalmente e de forma resumida, os componentes básicos de um diagrama Xchart (estados, transições, regras, eventos, ações, variáveis, atividades, etc) [16].

3.2.1 Estado

Um estado representa uma situação conceitual, contexto ou modo. Todos os recursos da linguagem Xchart estão associados a estados. Visualmente, um estado é representado por um retângulo com os cantos arredondados e que possui um identificador em seu interior. Um estado pode ser refinado em subestados, formando uma hierarquia de inclusão cujo identificador do estado raiz também identifica o Xchart que contém esta hierarquia. Alguns estados podem necessitar de recursos para controlar o volume de detalhes em um determinado nível da hierarquia. Um estado que contém o símbolo ∇ é descrito em separado. Nessa descrição adicional, o símbolo Δ indica que esse estado não é um novo Xchart, mas sim, o refinamento de um estado que é apenas referenciado em outra hierarquia. Um estado pode ser básico, exclusivo, concorrente ou fictício. Um estado que não possui subestados é dito básico. Os estados que possuem subestados podem ser concorrentes ou exclusivos. Um estado é exclusivo quando apenas um de seus subestados pode estar ativo em um determinado instante. Quando todos os subestados estão ativos simultaneamente, o estado é dito concorrente. Os estados exclusivos são representados por retângulos com contorno contínuo. Os estados concorrentes são representados por retângulos com contorno tracejado. Os estados fictícios são utilizados como origem de transições iniciais ou destino de transições finais. Esses estados são representados por um círculo (•).

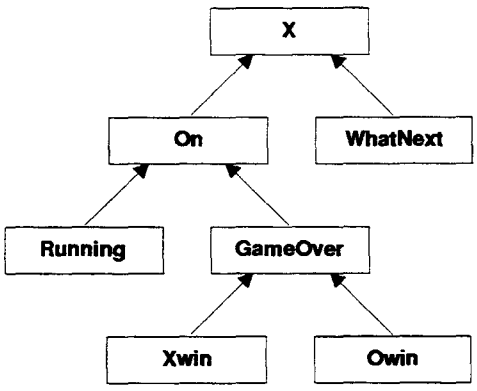


Figura 10 - Hierarquia de estados do exemplo da Figura 12

No exemplo da Figura 12, o Xchart **X** é composto pela hierarquia de estados mostrada na Figura 10. Nessa hierarquia, os estados ***WhatNext, Running, Xwin*** e ***Owin*** são básicos. Os estados ***X, On*** e ***GameOver*** são exclusivos. Nessa hierarquia não existe nenhum estado

concorrente. Adicionalmente, podem ser identificados quatro estados fictícios: o primeiro, utilizado como origem da transição que ativa o estado ***On*** (rotulada pelo evento ***turnX***), o segundo, utilizado como destino da transição pseudo-final cuja origem é o estado ***On*** (rotulada pelo evento ***turnO***) e os outros dois, utilizados como destino de transições finais cujas origens são o estado **X** (uma rotulada pelo evento ***gaveUp*** e a outra rotulada pela regra ***exit:raise(gaveUp,"O")***. Através dos controles de volume (∇ e Δ), o estado ***On*** é referenciado no diagrama Xchart **X** e descrito em detalhes separadamente.

3.2.2 Transição

Uma transição (→) estabelece uma relação entre estados. Esse é o único mecanismo através do qual estados podem ser ativados/desativados. Toda transição possui uma origem, um rótulo, um destino e um número de prioridade (opcional). Para uma transição ser executada, é necessário (mas não suficiente) que os estados origem estejam ativos e que a regra que a rotula esteja habilitada. Um conjunto de transições que satisfaz essas condições pode conter transições excludentes e, nesse caso, nem todas serão executadas. A execução de uma transição implica em: (1) desativar os estados origem, (2) executar a regra que rotula a transição e (3) ativar os estados destino. A ordem em que os estados origem/destino são desativados/ativados é determinada pela noção de Ancestral Comum Mais Próximo (ACMP) [15]. As transições podem ser classificadas em:

- Convencionais - estabelecem relações entre estados.
- Iniciais – têm como origem um estado fictício (•).
- Finais – têm como destino um estado fictício.
- Pseudo-finais – têm como destino um estado fictício no interior de um estado exclusivo.
- Internas - não provocam a desativação do(s) estado(s) origem.
- Externas - provocam a desativação de todos os subestados de um estado.

No exemplo da Figura 12, podem ser identificadas duas transições iniciais (uma ativa o estado ***On*** e a outra ativa o estado ***Running***), duas transições finais (ambas desativam o estado **X**), uma transições pseudo-final (desativa o estado ***On***), duas transições convencionais (ambas com origem em ***Running*** e destinos em ***Xwin*** e ***Owin*** respectivamente), uma transição interna (de **X** para ***WhatNext***) e duas transições externas (uma de ***On*** para **X** e a outra de ***WhatNext*** para **X**).

3.2.3 Evento

Evento primitivo é a abstração empregada para modelar um acontecimento, por exemplo, ligar um carro. Os eventos primitivos são instâncias de tipos de eventos primitivos. Esses tipos podem estar organizados em uma hierarquia de especialização. Os eventos primitivos são criados pelo ambiente externo ou por instâncias de Xcharts. Os tipos de eventos primitivos fazem parte da especificação.

Conceitualmente, um evento primitivo representa um acontecimento instantâneo, não persistente, ocorrido em um instante particular. A ocorrência de um evento primitivo é registrada através de um estímulo externo. A ocorrência de um evento primitivo implica na ocorrência do tipo de evento primitivo do qual ele é uma instância, além dos tipos que são seus ancestrais na hierarquia de tipos de eventos primitivos. Um evento primitivo pode carregar dados consigo, podendo ser visto como um veículo de transporte de informações entre instâncias ou entre instâncias e os demais subsistemas. Os dados são fornecidos pelo construtor do evento primitivo. As instâncias de Xcharts não manipulam os dados associados a um evento primitivo. Esses dados são utilizados exclusivamente por atividades.

Evento é uma construção mais sofisticada que eventos primitivos, elaborada a partir de uma possível combinação de vários elementos. Os elementos que podem compor um evento e as condições nas quais se diz que eles ocorrem são definidos abaixo:

- Evento nulo - ocorre quando qualquer evento primitivo ocorre.
- Tipo de evento primitivo - também pode ser caracterizado como um evento.
- Eventos associados a condições - ocorrem sempre que o valor de uma condição é alterado.
- Evento associado a expressões - ocorre sempre que o valor de uma expressão lógica ou numérica é alterado.
- Eventos associados a estados - ocorrem sempre que um estado é ativado ou desativado.
- Eventos associados a atividades - ocorrem sempre que uma atividade é iniciada ou finalizada.
- Eventos temporais - a cada estado está associado um cronômetro que gera os eventos temporais: *at*, *after* e *every*.
- Eventos compostos - eventos formados pela combinação de outros eventos através de operadores lógicos.

Os eventos são interceptados por regras no interior de estados ou regras que rotulam transições. Assim como a troca de mensagens em um sistema orientado a objetos determina o comportamento desse sistema, os eventos determinam o comportamento do controle descrito em Xchart.

3.2.4 Condição

Condição é uma expressão lógica cujo valor pode ser falso ou verdadeiro. As condições válidas em Xchart são:

- Condição nula - condição que sempre é satisfeita, seu valor é sempre verdadeiro.
- Condição associada a estado - condição verdadeira se o estado em questão está ativo.
- Condição associada a evento não ocorrido no passo corrente - verdadeira se o evento não faz parte do estímulo externo que iniciou o passo e também não foi gerado até o micropasso no qual a condição é avaliada.
- Condição associada ao valor de expressão no início de um passo - retorna o valor de uma expressão no início do passo, mesmo que micropassos anteriores tenham alterado esse valor.
- Condições associadas a atividades - essas condições são verdadeiras se uma atividade está ativa ou suspensa.
- Condições envolvendo expressões numéricas - esse tipo de condição é resultado da comparação de expressões numéricas através dos operadores =, <, >, etc.
- Condições compostas - resultado da combinação das condições anteriores através de operadores lógicos.

3.2.5 Ação

Ação é o mecanismo utilizado para gerar controle. Toda e qualquer reação de uma instância é expressa através de ações que alteram os valores de variáveis de controle ou manipulam atividades. Uma ação pode ser composta por vários elementos de ação que, em geral, são indivisíveis. A exceção fica por conta dos elementos de iteração. Qualquer seqüência de elementos de ação pode se tornar indivisível aplicando-se o modificador *atomic*. Os principais elementos de ação estão listados abaixo:

- Ação nula - não produz controle.
- Atribuição - atribui um valor, resultado da avaliação de expressões numéricas ou lógicas, a uma variável.
- Geração de eventos primitivos - a função *raise* é utilizada para gerar eventos primitivos. A visibilidade do evento primitivo é definida pelo tipo de chamada dessa função. O evento primitivo pode ser visível na instância corrente, em todas as instâncias de um determinado Xchart, em todas as instâncias do subsistema descrito em Xchart ou apenas em um estado de uma instância específica. Os eventos primitivos gerados em um passo tornam-se visíveis para as demais instâncias após o fim da execução desse passo. Os eventos primitivos gerados para a instância corrente, juntamente com o estímulo externo que iniciou a execução do passo, formam o estímulo externo acumulado, utilizado para identificar o próximo micropasso a ser executado. Não é possível gerar mais de um evento primitivo de mesmo tipo no mesmo micropasso.
- Espera por tipos de eventos primitivos - a instância que executa essa ação torna-se inoperante até que receba o evento primitivo do tipo especificado.
- Controle de atividades - ações que determinam quando uma atividade deve ser iniciada (de forma síncrona ou assíncrona), finalizada, suspensa, ou reiniciada (após ser suspensa). Estímulos externos sinalizados durante a execução síncrona de uma atividade são ignorados.
- Criação de instâncias de Xcharts - existem duas ações para criar instâncias de Xcharts. A diferença entre elas está no status resultante da instância que executa esse tipo de ação: essa instância pode se tornar inoperante enquanto a instância criada não for finalizada ou pode ser executada concorrentemente.
- Repetição - *while do* e *for* são estruturas utilizadas para executar repetidamente uma seqüência de ações. Essa seqüência não é indivisível (a menos que o modificar *atomic* seja utilizado).
- Decisão - decide se uma seqüência de ações deve ser executada de acordo com a avaliação de uma condição.

3.2.6 Regra

Uma regra é a união de um gatilho e de uma ação. Um gatilho é a combinação de um evento e de uma condição. Um gatilho é dito habilitado se, e somente se, o evento ocorre e o valor da condição é verdadeiro. As regras rotulam transições ou são especificadas no

interior de estados. Uma regra é executada no máximo uma vez por passo. Diz-se que uma regra está habilitada se o seu gatilho está habilitado. Uma regra habilitada não necessariamente é executada.

Para que um evento *E* seja interceptado por uma regra *R*, três condições devem ser satisfeitas: (1) o gatilho de *R* deve incluir o evento *E*; (2) a condição desse gatilho deve ser verdadeira; (3) o estado que contém *R* deve estar ativo ou, no caso de *R* rotular uma transição, os estados origem da transição devem estar ativos. Quando as duas primeiras condições são satisfeitas, diz-se que a regra está habilitada (pronta para ser executada). Quando as três condições são satisfeitas, diz-se que a regra (ou a transição rotulada pela regra) é relevante. Somente as regras/transições relevantes são executadas. Se a condição de uma regra é nula, então a regra sempre estará habilitada quando o evento ocorrer.

A hierarquia e os tipos dos estados determinarão a ordem de execução das regras (caso elas não sejam executadas concorrentemente). As regras dos estados ancestrais são executadas antes das regras dos subestados. Se não há relação entre dois estados, as regras desses estados são executadas concorrentemente. As regras que pertencem ao mesmo estado não são executadas concorrentemente. Uma ordem arbitrária ou a prioridade de cada uma irá estabelecer a ordem de execução.

3.2.7 Variável de controle e atividade

Uma variável de controle é um repositório que armazena um valor conforme o seu tipo. As variáveis, em relação a um Xchart, podem ser locais ou globais.

As atividades fornecem a funcionalidade do sistema cujo controle é modelado em Xchart. Podem ser implementadas como procedimentos, rotinas, métodos ou funções, em linguagens de programação convencionais. As atividades realizam operações além dos propósitos de Xchart. No entanto, o subsistema reativo oferece recursos que podem ser usados pelas atividades, por exemplo, a capacidade de acoplar dados aos eventos primitivos e recuperar esses dados. As atividades também podem gerar eventos primitivos, criar instâncias de Xcharts e manipular variáveis de controle globais. Essas operações são realizadas através da IPX (Interface de Programação de Xchart). Em um determinado instante, uma atividade pode estar inerte, ativa, ou suspensa.

3.3 Exemplo

A seguir, será apresentado um exemplo [20] de uma especificação em Xchart onde poderão ser identificados os principais recursos dessa linguagem. Será descrito o controle

de diálogo da interface de um sistema que implementa o tradicional "jogo da velha" (Figura 12). Uma partida do "jogo da velha" é realizada entre dois jogadores representados pelos símbolos **X** e **O**. Os jogadores escolhem, alternadamente, uma posição em um tabuleiro semelhante ao da Figura 11. O símbolo associado ao jogador (**X** ou **O**) é utilizado para preencher a posição escolhida. O jogo termina quando algum dos jogadores consegue distribuir seus símbolos em três posições consecutivas do tabuleiro (na horizontal, vertical ou diagonal). Nesse caso, o jogador é o vencedor da partida. Quando todas as posições do tabuleiro forem preenchidas e a situação anterior não for alcançada, é declarado o empate entre os jogadores.

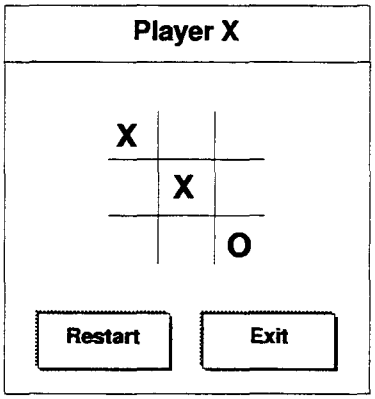


Figura 11 - Interface do jogador X em um determinado instante

O início da partida requer a existência de dois jogadores. Cada um desses jogadores visualiza sua própria interface (Figura 11). O jogador **X** inicia a partida escolhendo com o *mouse* uma posição no tabuleiro. Quando o *mouse* passa sobre uma posição desocupada, uma marca cinza preenche a posição. O jogador adversário não percebe tais movimentos. Caso o botão do *mouse* seja pressionado sobre uma posição inválida, uma campainha soa indicando uma exceção. Quando o botão é pressionado sobre uma posição vazia, o símbolo **X** preenche definitivamente a posição e o controle é transferido para o jogador adversário (**O**). O controle é alternado entre os jogadores enquanto o jogo não termina. Se não é a vez de um jogador, então nenhum *feedback* é oferecido quando esse jogador movimenta o *mouse*. Quando o jogo termina, uma pequena animação é executada: uma para o vencedor, parabenizando-o, e outra estimulando o jogador derrotado. Se a partida termina empatada, então uma mesma animação é exibida para ambos. Essas animações persistem até que os jogadores recomecem uma nova partida ou finalizem a execução do jogo. A finalização de

uma partida pode ocorrer em qualquer momento, por iniciativa de qualquer jogador. O reinício exige um consenso entre ambos.

O sistema que implementa o jogo é composto pelo **gerenciador de diálogo** (descrito em Xchart), por um cliente denominado ***Application*** (compartilhado entre os jogadores) e um par de clientes ***Animation*** e ***Interaction*** para cada jogador. O cliente *Animation* é responsável pelas animações produzidas pelo sistema. O cliente *Interaction* recebe as entradas do jogador, mapeia essas entradas para eventos em Xchart, fornece *feedback* imediato para algumas ações do jogador e executa atividades como, por exemplo, o *beep* (campainha). Cada jogador interage com os seus próprios clientes *Animation* e *Interaction.* Esses clientes sinalizam eventos para o gerenciador de diálogo via IPX. O cliente *Animation* recupera reações na porta **a** (**aX** para o cliente do jogador **X** e **aO** para o cliente do jogador **O**). O cliente *Interaction* recupera as reações na porta **i** (**iX** para o cliente do jogador **X** e **iO** para o cliente do jogador **O**). O cliente *Application* é responsável pela semântica do jogo, mais especificamente, esse cliente é responsável por armazenar o status de cada uma das posições do tabuleiro e indicar o jogador que fará a próxima jogada. O cliente *Application* sinaliza eventos para o gerenciador de diálogo e recupera as reações na porta **App**. A organização do código dos clientes está além dos interesses e restrições de Xchart. As funções do gerenciador de diálogo, bem como sua comunicação com os clientes serão descritas utilizando o ponto de vista do jogador **X.** O ponto de visto do jogador **O** é equivalente.

A especificação em Xchart do gerenciador de diálogo referente à interface do jogador **X** é mostrada na Figura 12. Quando uma instância do Xchart **X** é criada, o subestado ***On*** não é ativado. Esse estado é ativado somente quando o evento ***turnX*** é sinalizado (após a criação da instância). Esse evento é sinalizado pelo cliente *Application* (através da IPX) quando há outro jogador (no caso, **O**) interessado em jogar. Quando o estado ***On*** é ativado pela primeira vez, o evento ***turnX*** ainda habilita o jogador **X** a realizar sua primeira jogada. O estado ***On*** permanece desativado quando a jogada corrente é a do adversário e, nesse caso, a ocorrência de qualquer evento que não seja o evento ***turnX*** dispara uma campainha (conforme regra no interior de **X**), informando que a ação do jogador não é permitida no momento. O jogador poderia, por exemplo, estar tentando ocupar uma posição vazia durante a jogada do adversário. O estado ***On*** é desativado pelo evento ***turnO***, gerado pelo cliente *Application.* Quando o jogador **O** realiza o seu movimento, o cliente *Application* sinaliza novamente o evento ***turnX*** para a instância de **X**.

O estado ***On*** contém algumas regras. Uma delas é executada na ocorrência do evento ***mOverEmpty***, sinalizado pelo cliente *Interaction* quando o indicador do *mouse* passa sobre uma posição não preenchida do tabuleiro. Nesse caso, a reação é a execução síncrona da atividade ***DrawGray***. Essa reação é depositada na porta **iX** para ser tratada pelo cliente

Interaction. Se o evento gerado é ***cOverEmpty***, então jogador pressionou o botão do *mouse* sobre uma posição vazia do tabuleiro. A reação é preencher a posição com um **X** em preto e requisitar ao cliente *Application* (através da porta **App**) a avaliação da jogada. A jogada pode resultar em empate ou em vitória de um dos jogadores.

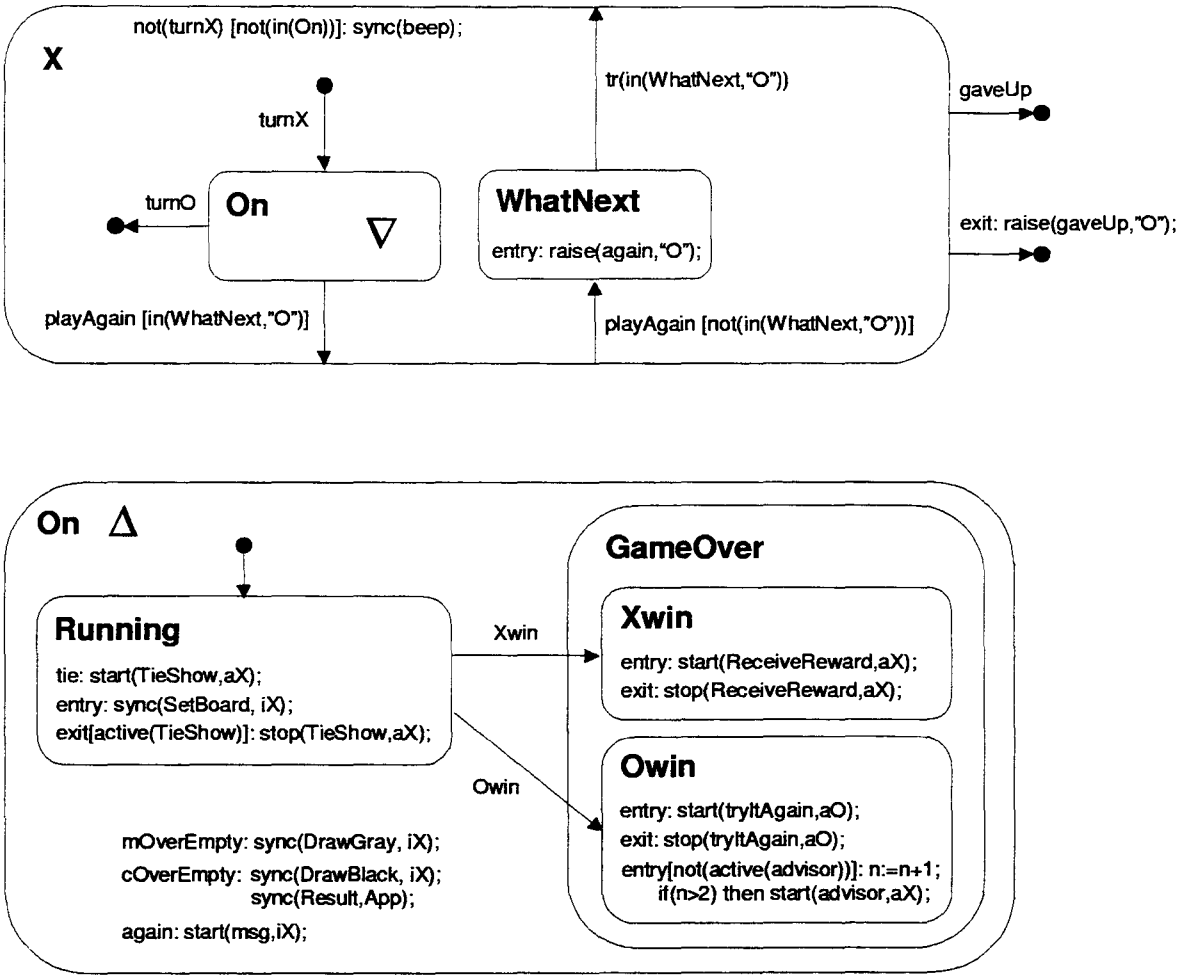


Figura 12 - Gerenciador de diálogo da interface do jogador X

O estado ***On*** agrega os subestados ***Running*** e ***GameOver***. Quando o estado ***On*** é ativado, o subestado ***Running*** é ativado em seguida, provocando a execução da atividade ***SetBoard***. Essa atividade exibe o tabuleiro em seu estado inicial, sem nenhuma marca. Como é de responsabilidade do cliente *Interaction*, essa atividade é depositada na porta **iX**. O estado ***Running*** também contém uma regra que trata os casos de empate. Se não há

vencedor, então o evento ***tie*** é sinalizado pelo cliente *Application* para ambos os jogadores. Ao receber o estimulo externo correspondente, a instância de **X** deposita a execução assíncrona da atividade ***TieShow*** na porta **aX** para ser tratada pelo cliente *Animation*. Ao desativar o estado ***Running***, a regra ***exit*** desse estado verifica se a atividade ***TieShow*** está em execução e, se for o caso, requisita a interrupção dessa atividade.

As transições rotuladas pelos eventos ***Xwin*** e ***Owin*** (partindo do estado ***Running***) capturam, respectivamente, a vitória do jogador **X** ou a vitória do jogador **O**. Essas transições têm destinos explícitos no interior do estado ***GameOver***. Os eventos ***Xwin*** e ***Owin*** são sinalizados pelo cliente *Application* durante a execução da atividade ***Result*** (executada ao término de cada jogada). A ativação de um dos subestados de ***GameOver*** estabelece o *feedback* a ser apresentado a cada jogador. Se o vencedor é o jogador **X**, então a atividade ***ReceiveReward*** é executada de forma assíncrona pelo cliente *Application*. A instância pode continuar reagindo a estímulos a medida que o cliente *Animation* executa essa atividade. Se o botão *Exit* é pressionado, o evento ***Exit*** é sinalizado pelo cliente *Interaction* e a transição final do Xchart **X** é executada. Ao tratar esse evento, a instância de **X** sinaliza para a instância de **O** o evento ***gaveUp***, que também provoca a finalização da instância do Xchart **O**. Se o botão *Restart* é pressionado, então o evento ***playAgain*** é sinalizado, causando a interrupção da animação ***ReceiveReward*** e a desativação do estado ***GameOver***.

O evento ***playAgain*** provoca a ativação do estado ***WhatNext***. Esse estado só pode ser desativado se algum jogador deseja finalizar a aplicação ou quando o jogador adversário também deseja iniciar uma nova partida. Se o jogador **X** deseja iniciar uma nova partida e o estado ***WhatNext*** está ativo, então o evento ***again*** é sinalizado para a instância do Xchart **O**. O estado ***WhatNext*** permanecerá ativo até que o estado ***WhatNext*** da instância de **O** torne-se ativo, caracterizando o consenso capturado pelo evento ***tr***. Se o jogador **O** toma a iniciativa de iniciar a partida, então o estado ***X*** permanecerá temporariamente básico, conforme a transição interna do estado ***On*** para **X**. Em ambos os casos, se existe o consenso entre os jogadores, o estado **X** permanece temporariamente básico até que o cliente *Application*, através dos eventos ***turnX*** ou ***turnO***, decida quem irá iniciar a partida.

O evento ***again*** provoca a execução da atividade ***msg***. Essa atividade exibe uma mensagem informado que o adversário deseja jogar novamente. A regra no interior do estado ***On*** captura esse comportamento. No interior do estado ***Owin*** existe uma regra de entrada que conta quantas partidas foram perdidas pelo jogador **X**. A terceira derrota provoca a execução assíncrona da atividade ***advisor***. Essa atividade é executada pelo cliente *Animation* e pode corresponder a um agente que irá sugerir os próximos lances para o jogador **X**.

Capítulo 4

Arquitetura reflexiva para a linguagem Xchart

Os sistemas com arquitetura reflexiva são mais reutilizáveis que os sistemas tradicionais pois permitem a seus clientes intervir, de forma controlada e modular, em sua implementação. Essa intervenção permite criar variações semânticas ou otimizações da implementação. O conceito de reflexão computacional sugere um novo paradigma para o desenvolvimento de sistemas. Nesse novo paradigma, um sistema é decomposto em dois níveis: o nível base, que agrupa a implementação do sistema propriamente dita e o nível reflexivo, que agrupa o código implementacional ou reflexivo. Utilizando reflexão, os sistemas tendem a se tornar mais estruturados, diminuindo sua complexidade, facilitando sua compreensão e sua manutenção. Algoritmos de políticas (como distribuição, persistência e concorrência), bem como quaisquer outros algoritmos que não estejam diretamente relacionados ao domínio do sistema podem ser implementados no nível reflexivo sem a necessidade de alterações no código do nível base. Por causa da separação de domínios, a especificação original é preservada, permitindo habilitar/desabilitar o domínio reflexivo sempre que necessário. Essa separação também permite a compreensão de cada domínio de forma independente.

As vantagens associadas à reflexão computacional, motivaram o projeto de uma arquitetura reflexiva para a linguagem Xchart. Outro fator decisivo foi que, embora existam arquiteturas reflexivas para linguagens de programação de diversos paradigmas, não temos conhecimento de nenhum trabalho similar em linguagens orientadas a eventos, como é o caso de Xchart. O paradigma de eventos favorece a criação de uma arquitetura reflexiva comportamental. Nesse paradigma, a transferência de controle entre os níveis base e reflexivo pode ser realizada de forma natural, através da interceptação de eventos. Assim como a troca de mensagens em um sistema orientado a objetos determina o comportamento desse sistema, a sinalização e a interceptação de eventos determina o comportamento do

gerenciador de diálogo descrito em Xchart. No caso particular de Xchart, o sistema de execução (SE) sinaliza eventos sempre que a configuração de uma instância é alterada, por exemplo, quando estados são ativados/desativados, quando ocorrem alterações nos valores de expressões/condições, quando atividades são iniciadas/finalizadas, etc. Através da interceptação desses eventos, é possível monitorizar, de forma simples e completa, o comportamento de toda a especificação. Adicionalmente, podem ser estabelecidas condições nas quais ocorrerá a interceptação. No paradigma orientado a objetos, ao contrário, as facilidades para a interceptação de mensagens estão disponíveis apenas no nível do interpretador da linguagem. Nesse paradigma, as arquiteturas reflexivas precisam oferecer mecanismos adicionais que tratem explicitamente o problema da interceptação de mensagens.

4.1 Objetivos

Nossa arquitetura visa atingir dois objetivos básicos: (1) promover a separação de domínios através da organização da especificação em múltiplos níveis e (2) permitir a adaptação do comportamento e também da estrutura da especificação, de forma transparente, utilizando o nível-reflexivo. Uma observação importante é que, em ambos os casos, o código descrito no nível reflexivo poderia ser descrito diretamente no nível base, utilizando apenas os recursos disponíveis na linguagem Xchart. Conforme discutido, a vantagem em utilizar o nível reflexivo é conceitual e diz respeito à organização da especificação.

A decomposição em níveis, de acordo com os múltiplos domínios envolvidos, é uma alternativa para a organização da especificação que diminui sua complexidade, facilita sua compreensão e sua manutenção e aumenta sua robustez. Nesse tipo de organização, cada domínio pode ser analisado, compreendido e mantido de forma mais modular, facilitando a intervenção de especialistas que, na maioria dos casos, não precisam conhecer todos os domínios que compõem um sistema. Quando o nível reflexivo é utilizado para promover a separação de domínios, não são previstas, nesse nível, operações de exclusão de elementos do nível base. Caso o nível reflexivo eliminasse elementos do nível base, o domínio descrito no nível reflexivo estaria intervindo no domínio descrito no nível base, contrariando os objetivos da separação de domínios. Nesse contexto, o nível reflexivo apresenta um caráter permanente. Permanente no sentido de que, em geral, o nível reflexivo não será habilitado/desabilitado com muita freqüência (a não ser em situações especiais, por exemplo, para depuração).

Muitas vezes, o programador é forçado a intervir na especificação, em caráter temporário, para realizar algum tipo de teste ou computação adicional. Durante as etapas de

testes, depuração e otimização da especificação, esse tipo de intervenção é comum. A intervenção direta na especificação compromete sua robustez, pois corre-se o risco de alterar ou desabilitar, acidentalmente, elementos indesejados. Caso não haja cópias da especificação original ou uma documentação bastante detalhada, a correção dos eventuais erros introduzidos torna-se penosa. Em descrições complexas, esses problemas se agravam ainda mais. Outro problema é que ao desfazer as alterações, essas alterações são perdidas, a menos que seja feita uma cópia do sistema adaptado. Caso seja necessário refazer as alterações, além do trabalho adicional, a robustez do sistema será novamente comprometida. Nesse novo contexto, o nível reflexivo pode ser utilizado para realizar intervenções na especificação de forma transparente. Todas as alterações no nível base passam a ser realizadas indiretamente, através do nível reflexivo. Assim, a especificação original (nível base) é preservada, garantindo a sua robustez. Por estarem agrupadas em um único nível, independente da especificação original, as alterações podem ser facilmente identificadas, compreendidas e desabilitadas. O nível reflexivo também pode ser preservado ao ser desabilitado. Dessa forma, é possível desfazer/refazer as alterações várias vezes, simplesmente habilitando/desabilitando o nível reflexivo. Quando utilizado com esse objetivo, as operações de remoção e substituição de elementos do nível base são comuns e o nível reflexivo apresenta um caráter temporário. Caso as adaptações fossem permanentes, não faria sentido utilizar o nível reflexivo. As adaptações deveriam ser realizadas diretamente no nível base, pois passariam a fazer parte da especificação.

4.2 A arquitetura

Nossa arquitetura reflexiva procura atingir os objetivos descritos na seção anterior preservando ao máximo as características da linguagem Xchart. Foram criadas três novas estruturas para a linguagem Xchart: (1) diagrama reflexivo; (2) elemento de ação *reuse*; (3) elemento de ação *remove*. Conforme será mostrado, essas três estruturas são simples e podem ser tratadas exclusivamente em tempo de compilação. Assim, em relação ao ambiente Xchart, apenas o compilador TeXchart precisa ser adaptado. O sistema de execução de Xchart (mais especificamente, o núcleo reativo) permanece inalterado. De acordo com o modelo reflexivo, os elementos de ação *reuse* e *remove* não seriam necessários explicitamente. Seria função do Sistema de Execução de Xchart (SE) realizar essas operações de forma transparente (a operação a ser realizada seria uma das metainformações disponíveis quando um evento fosse interceptado). Resolvemos adicionar, a principio, os elementos de ação *reuse* e *remove* à linguagem (relaxando o modelo reflexivo) por dois motivos: (1) tornar mais claras as operações associadas a esses elementos e (2) facilitar uma implementação inicial da arquitetura (em tempo de

compilação). A medida que nossa arquitetura for estendida e passar a ser tratada em tempo de execução (seção 5.2.2), inevitavelmente, esses elementos de ação deixarão de existir explicitamente na linguagem Xchart.

4.2.1 Diagrama reflexivo

Para promover a organização da especificação em níveis, foi criado um novo tipo de diagrama para a linguagem Xchart, denominado diagrama reflexivo. Um diagrama reflexivo pode ser considerado uma especialização de um diagrama Xchart. A principal diferença entre esses dois tipos de diagramas é a restrição de que um diagrama reflexivo deve estar associado a pelo menos um diagrama Xchart, ao qual estende, ou então, a outro diagrama reflexivo. Para diferenciar os dois tipos de diagramas, adotaremos a convenção de que, visualmente, um diagrama reflexivo apresenta um preenchimento cinza. A associação entre dois diagrama reflexivos ou entre um diagrama reflexivo e um diagrama Xchart é representada por uma linha unindo os diagramas.

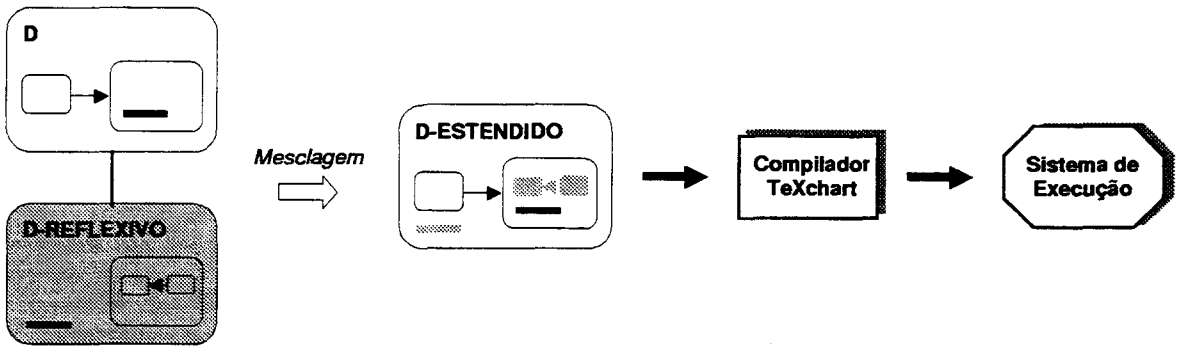


Figura 13 – Uma abordagem simples para a mesclagem de diagramas

O domínio relativo ao sistema propriamente dito deve sempre ser modelado utilizando-se diagramas Xchart. Esse domínio sempre corresponde ao nível base. Os demais domínios (ou extensões do nível base) devem ser descritos utilizando-se diagramas reflexivos. A arquitetura proposta é parcialmente reflexiva pois um diagrama Xchart pode ter ou não um diagrama reflexivo associado a si. Utilizando nossa arquitetura, é possível adicionar, remover ou alterar regras, transições e estados em qualquer um dos níveis da hierarquia de estados de um diagrama Xchart. O estado raiz da hierarquia é o único estado cujo identificador no diagrama reflexivo pode ser diferente do identificador no diagrama Xchart.

Essa exceção é necessária porque o estado raiz também é utilizado para identificar o diagrama no qual a hierarquia está contida. A possibilidade de associar um diagrama reflexivo a múltiplos diagramas Xchart (com identificadores distintos) também impede a utilização do mesmo identificador no diagrama reflexivo. No caso do estado raiz, a correspondência no nível reflexivo é determinada justamente por essa característica especial e não pela correspondência de identificadores. O diagrama reflexivo deve repetir os estados do diagrama Xchart que serão estendidos. No caso de extensões a um subestado da hierarquia, o diagrama reflexivo deve conter o ramo da hierarquia ao qual o subestado pertence, até o nível desse subestado. As regras e transições do diagrama Xchart não precisam ser repetidas no diagrama reflexivo, a não ser no caso em que esses elementos também sejam alterados.

A separação de domínios em diferentes diagramas existe apenas no nível da especificação. Em alguma etapa antes da execução, o código dos diagramas reflexivos é mesclado ao código dos diagramas Xchart aos quais estão associados. A mesclagem pode acontecer antes ou durante a etapa de compilação. O sistema de execução (SE) não precisa ser alterado porque o código executável é o mesmo da abordagem convencional (sem arquitetura reflexiva). Ao contrário de outras propostas (por exemplo, Open C++ [45]), não há *overhead* associado ao mecanismo de reflexão adotado. A abordagem mais simples para a implementação da arquitetura proposta é realizar a mesclagem das descrições textuais dos diagramas (em TeXchart) antes da etapa de compilação (Figura 13). Nessa abordagem, nem mesmo o compilador precisa ser alterado. Uma discussão mais detalhada sobre as diversas possibilidades para a implementação de nossa arquitetura será feita na seção 4.5.

Uma observação importante é que a responsabilidade pelo código descrito nos níveis reflexivos, assim como acontece quando os domínios estão todos misturados nos diagramas Xchart (abordagem convencional), é do programador. O processo de mesclagem detectará apenas erros relativos à referência em diagramas reflexivos de elementos inexistentes nos respectivos diagramas Xchart. Qualquer elemento descrito do nível reflexivo e que não faça parte do nível base será considerado uma extensão ao nível base e não ocasionará qualquer tipo de erro durante a mesclagem dos diagramas.

A seguir, serão apresentados alguns exemplos de diagramas reflexivos alterando diferentes características dos diagramas Xchart aos quais estão associados. Para facilitar a compreensão, em todos os exemplos, o diagrama Xchart é denominado **D**. O diagrama reflexivo associado ao diagrama Xchart é denominado **D-REFLEXIVO.** Com a intenção de esclarecer melhor como é feita a mesclagem desses diagramas, todo exemplo também irá

conter um diagrama denominado **D-ESTENDIDO**[2], que é o diagrama conceitual obtido após a mesclagem do diagrama reflexivo e do diagrama Xchart. A Figura 13 ilustra esse processo. Conforme pode ser observado, o código que é executado pelo SE é obtido da compilação de **D-ESTENDIDO** que, por sua vez, é obtido realizando-se a mesclagem dos diagramas **D** e **D-REFLEXIVO**.

4.2.2 Adição de regras

A Figura 14 mostra dois exemplos de como adicionar regras a um diagrama Xchart utilizando diagramas reflexivos. No exemplo (a), o diagrama Xchart **D** aparece vazio. O diagrama **D-REFLEXIVO** estende **D** adicionando a regra E[x<0]: x:=x+1;. O diagrama obtido após a mesclagem de **D** e **D-REFLEXIVO** é **D-ESTENDIDO**. Conforme comentado, **D-ESTENDIDO** não faz parte da especificação. Esse diagrama é utilizado nos exemplos apenas para esclarecer (em alto nível) como os diagramas **D** e **D-REFLEXIVO** serão executados. O exemplo (b) mostra a adição de uma regra no interior de um dos subestados do diagrama **D**. Observe que o subestado **B** e a transição de **A** para **B** não precisaram ser repetidos em **D-REFLEXIVO,** justamente porque esses elementos não sofreram qualquer alteração no nível reflexivo. O diagrama **D-REFLEXIVO** referenciou apenas o subestado **A**, adicionando a esse estado a regra E[x>1]: y:=TRUE;.

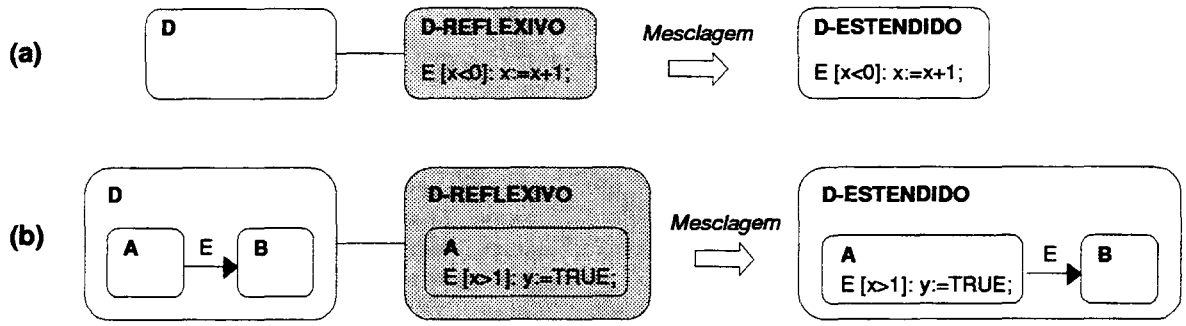


Figura 14 – Exemplos de adição de regras

[2] Na prática, o diagrama conceitual obtido após a mesclagem possui o mesmo identificador do diagrama Xchart.

4.2.3 Adição de transições e estados

A Figura 15 mostra um exemplo de como utilizar um diagrama reflexivo para adicionar transições e estados a um diagrama Xchart. O diagrama **D-REFLEXIVO** estende o diagrama **D** adicionando o estado **C** e a transição do estado **A** para o **C** (rotulada pelo evento E1). Essa transição é utilizada para ativar o estado **C**. Novamente, como não houve nenhum tipo de extensão ao estado **B** ou à transição de **A** para **B** (rotulada pelo evento E), esses elementos não precisaram ser referenciados em **D-REFLEXIVO**. Embora o estado **A** também não tenha sido estendido, o diagrama **D-REFLEXIVO** precisou referenciá-lo para que a transição de **A** para **C** pudesse ser completamente caracterizada no nível reflexivo.

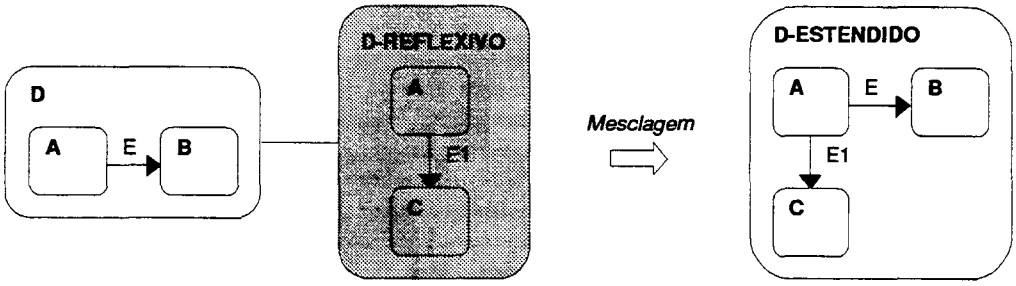


Figura 15 – Exemplo de adição de transição e estado

4.2.4 Elemento de ação *reuse*

Muitas vezes, o domínio descrito no nível reflexivo precisa adicionar alguns elementos de ação a uma regra já existente no nível base. Para simplificar esse tipo de operação, é utilizado um novo elemento de ação denominado *reuse*. *Reuse* é utilizado para reutilizar a ação de uma regra do nível base no nível reflexivo, evitando cópias redundantes dessa ação. Ao invés de repetir a ação no diagrama reflexivo, é possível reutilizar toda a ação da regra original referenciando-a através de *reuse*. Durante a mesclagem dos diagramas, *reuse* é substituído pela ação da regra à qual referencia. No nível reflexivo, é possível adicionar novos elementos de ação antes ou depois da ação referenciada. O procedimento para utilização de *reuse* é o seguinte: (1) repetir o gatilho da regra do diagrama Xchart que será estendida no diagrama reflexivo (dentro do estado correspondente); (2) referenciar a ação dessa regra utilizando *reuse*; (3) adicionar os novos elementos de ação antes e/ou depois de *reuse*. Caso o gatilho da regra seja repetido mas *reuse* não seja utilizado, a ação da regra do

nível base (diagrama Xchart) será substituída pela ação descrita no nível reflexivo (diagrama reflexivo).

A Figura 16 mostra dois exemplos de utilização de *reuse* para estender a ação de uma regra. No exemplo (a), **D-REFLEXIVO** adiciona o elemento de ação y:=0; ao final da ação da regra descrita no diagrama **D**. A utilização de *reuse* antes de y:=0; (em **D-REFLEXIVO**) determina essa posição. O exemplo (b) mostra o resultado da não utilização de *reuse* quando o gatilho de uma regra do nível base é referenciado no nível reflexivo. No exemplo, a regra E[x<0]: x:=x+1;, descrita no diagrama **D**, foi referenciada em **D-REFLEXIVO** sem a utilização de *reuse* (apenas o gatilho E[x<0] foi repetido). Nessas condições, a ação descrita em **D** (x:=x+1;) é substituida pela ação descrita em **D-REFLEXIVO** (x:=0;).

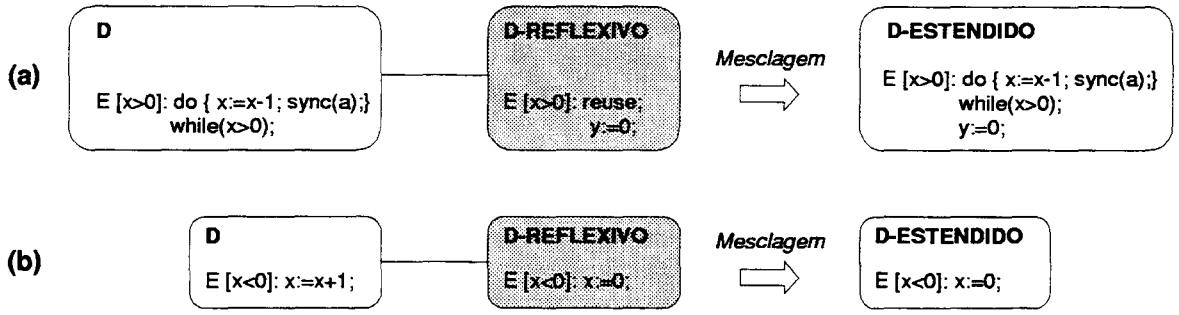


Figura 16 – Exemplos de alteração de regras

4.2.5 Elemento de ação *remove*

As estruturas descritas nas seções anteriores permitem estender diagramas Xchart utilizando diagramas reflexivos. Essas estruturas são suficientes para promover a separação de domínios em níveis distintos. No entanto, para realizar adaptações em um diagrama Xchart, além da capacidade de estendê-lo, deve ser possível também remover elementos desse diagrama. Para realizar esse tipo de operação, é preciso utilizar um novo elemento de ação denominado *remove*. Esse elemento é utilizado, basicamente, para remover uma regra descrita no nível base. A utilização de *remove* é simples. Para remover uma regra, basta referenciá-la no nível reflexivo (repetindo seu gatilho) substituindo a ação original pelo elemento de ação *remove*. Nesse caso, como a intenção é justamente substituir a ação original por *remove*, o elemento de ação *reuse* não precisa ser utilizado. Durante a

mesclagem dos diagramas, a presença de *remove* no nível reflexivo causa a eliminação da regra correspondente no nível base.

A remoção de regras pode ser utilizada para remover, indiretamente, transições e estados. Toda transição é rotulada por uma regra (mesmo que essa regra tenha evento, condição e ação nulos). Ao remover a regra que rotula uma transição, um dos elementos que caracterizam essa transição deixa de existir. Essa situação é utilizada, durante a mesclagem dos diagramas, para indicar a remoção da transição cuja regra foi eliminada. Durante a mesclagem são eliminadas apenas as transições cujas regras forem explicitamente removidas no nível reflexivo. A remoção de transições pode ser utilizada para remover estados. A remoção (no nível reflexivo) de todas as transições que ativam um estado implica em que esse estado jamais será ativado. Essa situação é um indicador para que esse estado seja removido durante a mesclagem dos diagramas.

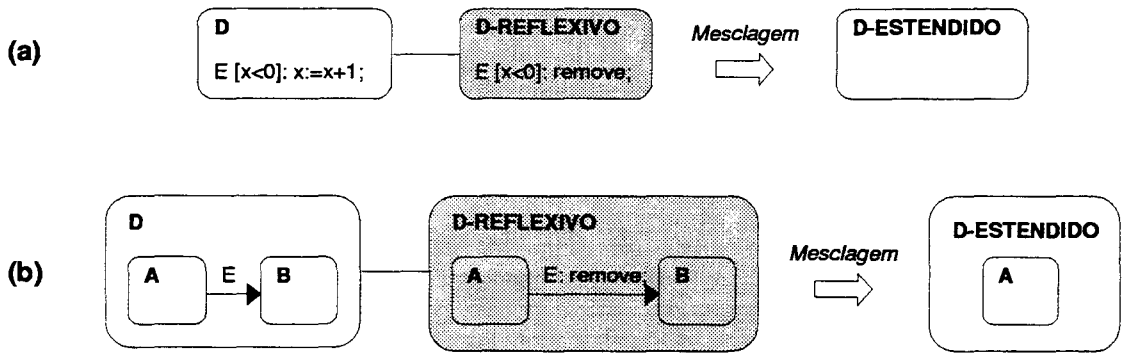


Figura 17 – Exemplos de remoção de regras, transições e estados

A remoção de uma transição ou de um estado pode provocar, implicitamente, a exclusão de outras transições e estados, ocasionando um efeito cascata. Considere, por exemplo, que a única transição que ativa um estado é removida. A remoção dessa transição implica em que o referido estado jamais será ativado e, por isso, também deve ser removido. A eliminação desse outro estado implica também em remover todas as transições cuja origem é esse estado (já que uma transição não pode existir sem uma origem). Essas transições eliminadas necessariamente ativam outros estados e, quando são as únicas a ativá-los, esses estados também devem ser removidos. Esse processo se repete enquanto houverem transições e estados que não puderem ser ativados/percorridos por causa das remoções realizadas. Deve-se observar que a eliminação de um estado implica na

eliminação de todos os seus subestados e também de todas as regras e transições descritas em seu interior.

A Figura 17 mostra dois exemplos da utilização de *remove*. No exemplo (a), a utilização de *remove* em **D-REFLEXIVO** elimina a regra descrita no diagrama **D**. No exemplo (b), a utilização de *remove* na regra que rotula a transição de **A** para **B** elimina essa transição. Os estados **A** e **B** foram repetidos em **D-REFLEXIVO** para que a transição pudesse ser completamente caracterizada. A remoção da transição implica também na remoção do estado **B**, já que essa era a única transição que o ativava. Assim, após a mesclagem dos diagramas (**D-ESTENDIDO**), apenas o estado **A** continua existindo.

4.3 Exemplos

A seguir, serão descritos dois exemplos de utilização da arquitetura reflexiva proposta. Ambos os exemplos utilizam a especificação do gerenciador de diálogo do jogo da velha, descrita na seção (3.3), como base para algumas extensões e adaptações utilizando o nível reflexivo. Nos dois exemplos, o diagrama **X** (mostrado em detalhes na Figura 12) será apenas referenciado. O controle de volume (∇) será utilizado para simbolizar que os detalhes desse diagrama estão descritos em outro local. O primeiro exemplo utiliza o nível reflexivo para descrever um novo domínio. Esse novo domínio é utilizado para calcular algumas estatísticas sobre a execução das instâncias criadas a partir do diagrama **X**. O segundo exemplo utiliza o nível reflexivo para realizar algumas adaptações temporárias na especificação original.

4.3.1 Separação de domínios

Nesse exemplo, o diagrama reflexivo (**X-reflexivo**) é utilizado para acrescentar um novo domínio ao diagrama Xchart **X**. O novo domínio é formado por estatísticas sobre a execução das instâncias de **X**. Para facilitar a compreensão, foram escolhidas estatísticas simples. O diagrama reflexivo adiciona regras para calcular o número de empates, vitórias e derrotas do jogador **X**. Diversas outras estatísticas poderiam ser calculadas, por exemplo, o tempo de cada partida, o tempo médio que o jogador **X** gasta em cada jogada, o número de "cliques" em posições inválidas do tabuleiro, etc. As variáveis *vtie*, *vXwin* e *vOwin* são utilizadas para armazenar, respectivamente, o número de empates, vitórias do jogador **X** e vitórias do jogador **O** (ou, em outras palavras, derrotas do jogador **X**). Essas variáveis são zeradas pela regra: entry: vtie:=0; vXwin:=0; vOwin:=0; sempre que o estado raiz é ativado. Embora essa regra esteja contida no estado **X-reflexivo** (estado raiz do diagrama reflexivo), ela refere-se ao estado raiz do diagrama Xchart. Após a mesclagem, essa regra estará

contida no estado **X** (raiz do diagrama Xchart). Essa é a única regra acrescentada pelo diagrama **X-reflexivo**. As demais regras já existem no diagrama **X** e são estendidas, através da adição de elementos de ação, no diagrama **X-reflexivo**.

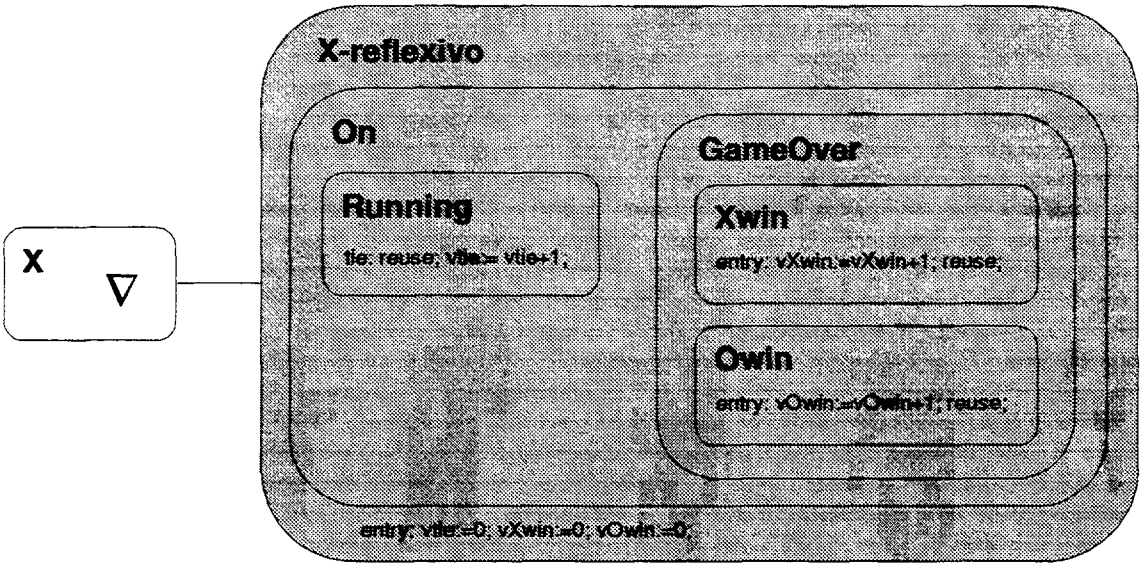


Figura 18 – Exemplo da separação de domínios utilizando o nível reflexivo

As regras no interior dos estados ***Xwin*** e ***Owin*** são utilizadas para atualizar, respectivamente, as variáveis *vXwin* e *vOwin*. O primeiro elemento de ação dessas regras incrementa a respectiva variável. Como essas regras já existem no nível base, para que a ação original seja executada, essa ação é referenciada pelo elemento de ação *reuse* após o incremento da variável. Embora não tenha sido realizada nenhuma extensão nos estados ***GameOver*** e ***On***, esses estados foram repetidos no diagrama reflexivo porque são ancestrais dos estados que foram estendidos (***Xwin*** e ***Owin***). A regra no interior do estado ***Running*** incrementa a variável *vtie* quando acontece um empate. Novamente, como essa regra já existe no nível base, sua ação é referenciada através de *reuse*. Ao contrário dos casos anteriores, a ação original será executada antes do incremento da variável (*reuse* precede o elemento de ação que incrementa *vtie*). O estado ***WhatNext*** não precisou ser repetido no diagrama reflexivo porque não foram realizadas alterações nesse estado. Também não foi necessário repetir as regras e as transições contidas nos estados referenciados porque esses elementos não foram alterados no diagrama reflexivo.

4.3.2 Intervenção no nível base

O diagrama reflexivo (**X-reflexivo**) mostrado na Figura 19 é utilizado para alterar, temporariamente, a especificação descrita no diagrama Xchart **X**. A alteração consiste em substituir o estado ***GameOver*** por regras no interior do estado ***Running***. O objetivo é verificar, em tempo de execução, o impacto dessas alterações em relação à performance. Para realizar essas alterações, foi necessário repetir no diagrama reflexivo todos os estados da sub-hierquia cuja raiz é o estado ***On***. O estado ***WhatNext*** não precisou ser repetido porque não sofreu nenhum tipo de alteração. Para remover o estado ***GameOver***, as transições do estado ***On*** para os subestados ***Xwin*** e ***Owin*** foram removidas. Essas são as transições que ativam o estado ***GameOver*** e a sua remoção implica na remoção do estado ***GameOver***, dos subestados ***Xwin*** e ***Owin*** e das regras contidas nesses estados. Para remover as transições, foi necessário remover as regras que as rotulam utilizando o elemento de ação *remove*.

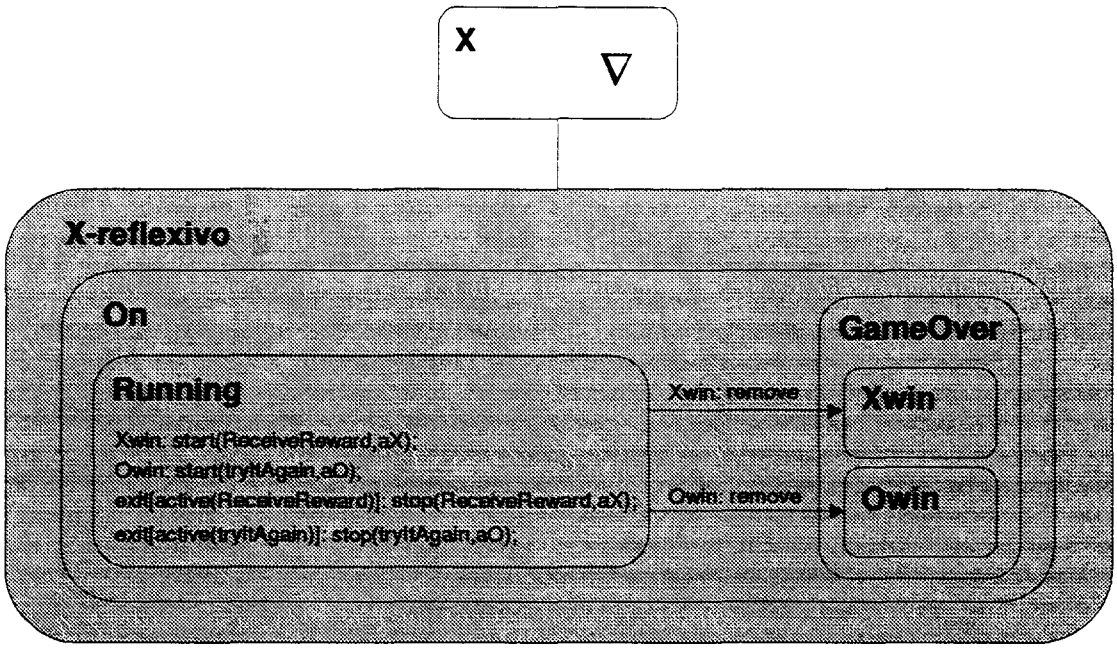


Figura 19 – Exemplo de intervenção no nível base utilizando o nível reflexivo

Ao remover o estado ***GameOver***, as regras no interior do estados ***Xwin*** e ***Owin*** também foram removidas. Essas regras são utilizadas para iniciar e finalizar as animações exibidas

aos jogadores quando o jogo não termina empatado. As novas regras para iniciar/finalizar essas animações foram adicionadas no interior do estado ***Running*** (com as devidas adaptações em relação às regras originais). No diagrama **X**, a vitória do jogador X é indicada pela sinalização do evento *Xwin*. Esse evento causava a transição do estado ***On*** para o estado ***Xwin*** que, ao ser ativado, iniciava a exibição da animação ***ReceiveReward***. Essa animação era finalizada quando o estado ***Xwin*** era desativado. No caso da vitória do jogador O, o processo era equivalente (utilizando o estado ***Owin***). No diagrama reflexivo, o evento ***Xwin*** passa a ser interceptado por uma regra no interior do estado ***Running***. A ação dessa regra dispara a animação ***ReceiveReward***. A animação é finalizada quando o estado ***Running*** é desativado. O tratamento da vitória do jogador O é equivalente.

4.4 Torre reflexiva

Os exemplos descritos nas seções anteriores associaram apenas um diagrama reflexivo a cada diagrama Xchart, embora essa não seja uma restrição. A relação entre diagramas Xchart e diagramas reflexivos é do tipo *m:n*. Um diagrama reflexivo pode estar associado a múltiplos diagramas Xchart, assim como um diagrama Xchart pode ter múltiplos diagramas reflexivos associados a si. Quando um diagrama reflexivo é associado a múltiplos diagramas Xchart (*m*:1), podem ser referenciados nesse diagrama apenas os elementos que sejam comuns a todos os diagramas Xchart envolvidos. Por exemplo, caso uma regra seja estendida no diagrama reflexivo, essa regra deve existir em todos os diagrama Xchart aos quais o diagrama reflexivo está associado. Por causa dessa restrição, em geral, o diagrama reflexivo é utilizado para adicionar novas regras no interior do estado raiz dos diagramas Xchart (todo diagrama Xchart contém ao menos o estado raiz). Esse tipo de relação (*m*:1) permite que o nível reflexivo seja utilizado não apenas para separar domínios, mas também para agrupar elementos comuns a diversos diagramas Xchart. Esse tipo de utilização diminui a complexidade da especificação na medida em que diminui, por exemplo, o número de réplicas de uma mesma regra nos diversos diagramas.

A associação de múltiplos diagramas reflexivos ao mesmo diagrama Xchart (1:*n*) forma uma torre reflexiva. Nessa torre, o primeiro diagrama reflexivo (r_1) é associado diretamente ao diagrama Xchart (x). Nesse caso, x é o nível base em relação a r_1. O segundo diagrama reflexivo (r_2) é associado ao diagrama reflexivo r_1, e assim sucessivamente. O *n*-ésimo diagrama reflexivo é associado ao diagrama r_{n-1}. Para o diagrama reflexivo r_n, o nível base corresponde à mesclagem do diagrama Xchart x com os n-1 diagramas reflexivos anteriores a r_n na torre reflexiva. A mesclagem dos diagramas é realizada de forma *bottom-up*. Inicialmente, o diagrama Xchart x é mesclado ao diagrama reflexivo r_1. O diagrama intermediário obtido dessa mesclagem (m_1) é então mesclado ao diagrama reflexivo r_2. Esse

processo continua até que o diagrama intermediário m_{n-1} seja mesclado ao diagrama reflexivo r_n. A Figura 20 mostra um exemplo do processo de mesclagem de uma torre reflexiva. Nesse exemplo, a primeira etapa do processo mescla o diagrama Xchart **D** com o primeiro diagrama reflexivo da torre, no caso, **D-REFLEXIVO1**. O diagrama obtido após essa primeira etapa é **D-ESTENDIDO1**. Conceitualmente, após a primeira etapa, os diagramas **D** e **D-REFLEXIVO1** são substituídos na torre reflexiva por **D-ESTENDIDO1**. O diagrama reflexivo **D-REFLEXIVO2** passa a estar associado a **D-ESTENDIDO1**. A segunda e última etapa do processo (já que a torre é composta de apenas dois diagramas reflexivos) mescla **D-ESTENDIDO1** com **D-REFLEXIVO2**, gerando o diagrama **D-ESTENDIDO2** que corresponde a toda a torre reflexiva. **D-ESTENDIDO2** é o diagrama que será efetivamente compilado e executado.

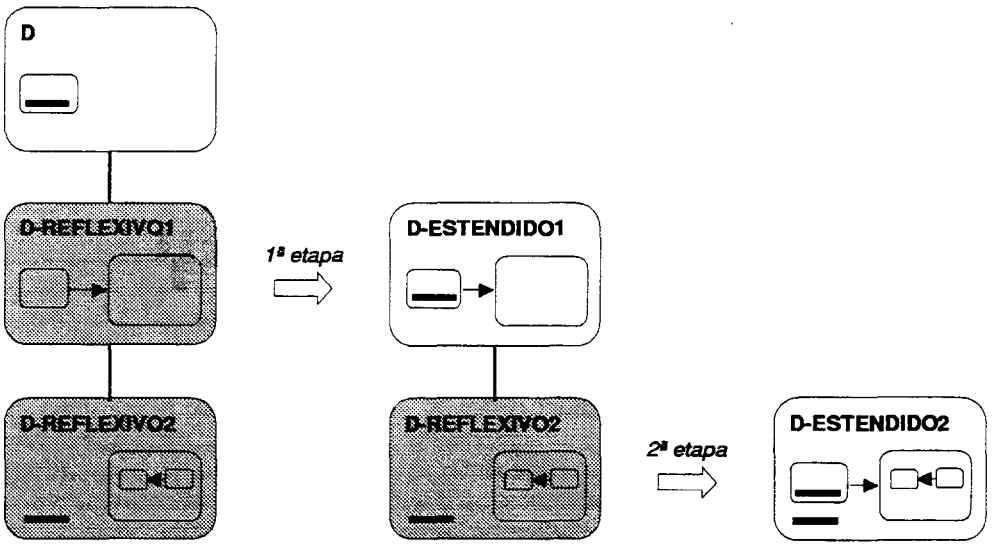


Figura 20 – Exemplo da mesclagem dos diagramas de uma torre reflexiva

Dada um torre reflexiva, em relação a um diagrama reflexivo *ri*, o nível base corresponde à mesclagem do diagrama Xchart com os *i*-1 diagramas reflexivos anteriores a *ri* na torre reflexiva. No diagrama *ri*, podem ser realizadas extensões aos elementos do diagrama Xchart e também em qualquer um dos *i*-1 diagramas reflexivos que precedem *ri* na torre. Por causa disso, a ordem do diagramas reflexivos na torre reflexiva deve ser tratada com cuidado. Diagramas reflexivos cujos elementos são referenciados por outro diagrama reflexivo devem precedê-lo na torre reflexiva.

Do ponto de vista da especificação como um todo (diagramas Xchart e torres reflexivas associadas), uma questão importante a ser considerada é a organização das múltiplas torres reflexivas. É desejável que haja correspondência entre os domínios e os níveis utilizados para descrever esses domínios em cada torre. Dessa forma, um especialista interessado em analisar o código relativo a um domínio específico analisa o mesmo nível em todas as torres da especificação. Essa organização também facilita a habilitação/desabilitação dos domínios temporários (utilizados para adaptar o nível base) de forma automática. O fato da arquitetura proposta ser parcialmente reflexiva torna o problema de garantir a correspondência entre os níveis mais complexo. Da mesma forma que um diagrama Xchart pode ter ou não um diagrama reflexivo associado, uma torre reflexiva pode conter ou não todos os níveis existentes nas outras torres da especificação. Assim, cada torre pode ter sua própria configuração em relação ao número de níveis (mas não em relação à ordem entre os níveis e à separação dos domínios). Torres que eventualmente não contenham algum nível intermediário devem mostrar um diagrama reflexivo vazio nesse nível para garantir que um determinado domínio estará no mesmo nível em todas as torres reflexivas da especificação. Quando os níveis ausentes são os últimos níveis da torre, não há a necessidade da utilização de diagramas vazios para completar a torre.

4.5 Implementação

A implementação da arquitetura proposta implica em alterações em dois componentes do ambiente Xchart: o editor de Xchart e o compilador TeXchart (para realizar a mesclagem dos diagramas). Conforme será discutido no próximo capítulo, algumas extensões imediatas de nossa arquitetura também irão requerer alterações no sistema de execução (SE) e na IPX. No caso dessas extensões, os processos de mesclagem e compilação dos diagramas (tal como será discutido a seguir) deverá ser adaptado.

4.5.1 Editor de Xchart

Os objetivos de nossa arquitetura são relativos à organização e à adaptação do gerenciador de diálogo descrito em Xchart. A eficácia da arquitetura deve ser testada pelos usuários de Xchart durante a codificação e/ou manutenção de uma especificação. Nesse aspecto, o editor de Xchart é indispensável. Na realidade, o editor de Xchart funciona como um complemento à nossa arquitetura. A organização da especificação em níveis possibilita oferecer ao usuário diferentes "visões" da especificação e avançados recursos de navegação. É possível oferecer ao usuário, por exemplo, a capacidade de visualizar cada domínio da especificação individualmente, habilitar/desabilitar domínios através de comandos simples,

exibir/ocultar diagramas, domínios e torres reflexivas. Somente com a existência de um editor que faça uso efetivo das possibilidades oferecidas pela arquitetura reflexiva proposta é possível realizar testes que comprovem sua eficácia.

4.5.2 Compilador TeXchart

A mesclagem dos diagramas Xchart e reflexivos pode ser realizada antes ou durante a compilação da especificação. A mesclagem da descrição textual dos diagramas (em TeXchart) antes da compilação é a abordagem mais simples. Nesse caso, nem mesmo o compilador TeXchart precisaria ser alterado.

No entanto, a abordagem ideal é realizar a mesclagem a medida que os diagramas são compilados. Nessa abordagem, o diagrama Xchart é compilado normalmente. Caso haja uma torre reflexiva associada a esse diagrama, a mesclagem dos diagramas reflexivos com o diagrama Xchart é realizada iterativamente, de forma *bottom-up* (em relação à ordem dos diagramas na torre). O primeiro diagrama da torre é mesclado ao diagrama Xchart. As operações de adição são realizadas primeiro porque essas operações não requerem nenhum tipo de tratamento adicional, a não ser nos casos em que os elementos do nível base são referenciados no nível reflexivo. Inicialmente, são adicionados os novos estados, em seguidas as novas transições e finalmente as novas regras. Os erros detectados pela mesclagem acontecem quando as regras descritas no diagrama reflexivo utilizam os elementos de ação *reuse* ou *remove* e não existem as regras correspondentes no diagrama Xchart. Os casos em que o usuário referencia estados ou transições inexistentes não geram erro. Nesses casos, assume-se que o diagrama reflexivo está adicionando esses elementos ao diagrama Xchart. Terminadas as operações de adição, são realizadas as operações de remoção. Essas operações devem ser iniciadas nos estados folha (da hierarquia de estados) e continuarem de forma *bottom-up*, dos subestados em direção ao estado raiz. Esse processo evita que sejam gerados erros quando o diagrama reflexivo exclui explicitamente um estado e também elementos no interior desse mesmo estado. A exclusão de transições e estados pode gerar um efeito em cascata, ocasionando indiretamente a exclusão de outras transições/estados. Esse efeito em cascata é tratado após serem realizadas todas as remoções que estão explícitas no diagrama reflexivo. A partir daí, o grafo que representa a relação entre estados e transições é analisado para tratar o efeito cascata das exclusões. Nesse grafo, os nós representam os estados. As arestas representam as transições entre esses estados. Após terminada a mesclagem com o primeiro diagrama reflexivo, o processo repete-se, de forma *bottom-up*, com os demais diagramas da torre reflexiva.

Algumas extensões imediatas que irão implicar em grandes mudanças na implementação da arquitetura proposta (discutidas no próximo capítulo), assim como a

impossibilidade de realizar testes que comprovem a eficácia dessa arquitetura sem utilizar um editor de Xchart, nos levaram a escolher, a princípio, a opção mais simples para a implementação. A mesclagem das torres reflexivas acontece antes da compilação da especificação, utilizando a descrição textual dos diagramas (em TeXchart). Essa implementação é temporária e tem por objetivo, simplesmente, viabilizar a utilização de nossa arquitetura.

Capítulo 5

Conclusões e trabalhos futuros

5.1 Conclusões

5.1.1 Implementação Aberta e Reflexão Computacional

Sistemas com implementação aberta são mais reutilizáveis que aqueles que se comportam como caixas-pretas pois permitem aos clientes para os quais a interface ou a estratégia de implementação do sistema não é otimizada intervir, de forma controlada, em sua implementação. O acesso à implementação é realizado através de uma metainterface. Uma metainterface é uma segunda interface adicionada ao modelo de caixa-preta, independente de sua interface original e com objetivos distintos. O código do cliente relativo à metainterface é denominado código implementacional e pode ser declarativo ou programável. As metainterfaces podem oferecer três níveis de acesso à implementação: (1) instrospecção, (2) invocação e (3) intervenção. Os níveis de introspeção e invocação são utilizados quando a interface do sistema oculta informações de estado ou funcionalidades que podem interessar a algum perfil de cliente. O nível de intervenção é o mais elaborado e deve ser utilizado quando a estratégia da implementação deve ser adaptada para satisfazer a novos requisitos de performance ou então oferecer alguma variação semântica. O tema de uma metainterface diz respeito às características da implementação às quais essa metainterface se refere explicitamente. Três exemplos de temas são: (1) o perfil do cliente; (2) a performance requerida pelo cliente; (3) a estratégia da implementação. A escolha do tema apropriado é uma decisão importante no projeto de metainterfaces pois afeta a facilidade de desenvolvimento, especificação, implementação e utilização dessas metainterfaces. Metainterfaces com o mesmo tema podem ter diferentes estilos. Por

exemplo, dois possíveis estilos para uma metainterface cujo o tema é a estratégia para a implementação são: (1) permitir ao cliente escolher uma das estratégias disponíveis diretamente ou (2) permitir ao cliente fornecer uma nova estratégia para a implementação. Temas e estilos diferentes podem ser combinados na mesma metainterface através da técnica de *layering*. Alguns princípios que devem guiar o projeto de metainterfaces são: (P1) devem ser expostas apenas informações essenciais sobre a implementação; (P2) deve haver uma separação clara entre o código que utiliza a interface do sistema e o código implementacional; (P3) o código implementacional deve ser opcional e fácil de desabilitar; (P4) o escopo de influência do código implementacional deve ser controlado de forma natural e suficientemente granular. O principal objetivo desses princípios é preservar o máximo possível as virtudes do modelo de caixa-preta.

A capacidade dos clientes de um sistema com implementação aberta intervirem em sua implementação é conhecida como reflexão implementacional (RI). RI é uma visão alternativa (e mais genérica) do conceito de reflexão computacional (RC). RC é a atividade realizada por sistemas reflexivos quando esses computam sobre si mesmos, alterando seu próprio comportamento, estrutura ou status. Sistemas reflexivos são sistemas computacionais que incorporam dados (denominados auto-representação) que representam aspectos estruturais e/ou comportamentais do próprio sistema. O modelo reflexivo sugere um novo paradigma para o desenvolvimento de sistemas. Nesse novo paradigma, um sistema é decomposto em pelo menos dois níveis: o nível base, que agrupa a implementação do sistema e o nível reflexivo, que agrupa o código implementacional ou reflexivo. Utilizando reflexão, os sistemas tendem a ser mais estruturados, diminuindo sua complexidade, facilitando sua compreensão e sua manutenção. Algoritmos de políticas, bem como quaisquer outros algoritmos que não estejam diretamente relacionados ao domínio do sistema podem ser implementados no nível reflexivo sem a necessidade de alterações no código do nível base. Sistemas que oferecem facilidades reflexivas permitem aos clientes consultar ou manipular alguns aspectos da implementação usando um conjunto predefinido de operações. Sistemas com arquitetura reflexiva permitem acesso mais amplo à implementação. Arquiteturas reflexivas são um caso particular de implementações abertas. De acordo com o tipo de auto-representação do sistema, as arquiteturas reflexivas podem ser classificadas em estruturais ou comportamentais. Uma auto-representação estrutural reifica aspectos estruturais de um programa. Uma auto-representação comportamental reifica o processo computacional e seu comportamento. Reificação é o processo de converter programas em dados. O processo inverso é denominado reflexão. O nível reflexivo de um sistema pode existir em tempo de execução ou em tempo de compilação. As arquiteturas que trabalham em tempo de execução introduzem *overhead* adicional ao sistema porque, para executar uma operação no nível base, deve ser executado o código no nível reflexivo que a implementa. Um forma de diminuir esse *overhead* é

utilizar a técnica de avaliação parcial. As arquiteturas que trabalham em tempo de compilação não introduzem *overhead* relativo ao mecanismo de reflexão. Nesse tipo de arquitetura, o código reflexivo modifica o compilador para que o código compilado se comporte de acordo com as intenções do usuário.

Uma característica importante de um sistema reflexivo é que deve haver uma relação causal entre esse sistema e sua auto-representação, ou seja, alterações no sistema devem implicar em alterações em sua auto-representação, e vice-versa. Grande parte das pesquisas em RC se utilizam de interpretadores meta-circulares para satisfazer essa relação causal. Um interpretador meta-circular é um interpretador escrito na linguagem que interpreta e que utiliza algumas formas especiais, adicionadas a essa linguagem, que permitem aos programas em execução ter acesso e modificar o estado de seu interpretador. Uma alternativa recente aos interpretadores metacirculares são os MOPs (protocolos de metaobjetos). Um MOP é uma metainterface orientada a objetos. Os objetos da metainterface que correspondem à auto-representação do sistema são denominados metaobjetos. De acordo com a relação objeto/metaobjeto, as arquiteturas reflexivas podem ser classificadas em parcialmente ou completamente reflexiva. Enquanto em arquiteturas completamente reflexivas existem metaobjetos associados a todos os objetos do nível base, em arquiteturas parcialmente reflexivas os metaobjetos são associados apenas aos objetos que sofrerão algum tipo de computação reflexiva.

5.1.2 Arquitetura reflexiva para a linguagem Xchart

Nossa motivação para o projeto de uma arquitetura reflexiva para a linguagem Xchart se baseou em dois fatores: as vantagens associadas ao modelo reflexivo e a inexistência de arquiteturas reflexivas em linguagens do paradigma de eventos. O paradigma de eventos favorece a criação de uma arquitetura reflexiva comportamental. Nesse paradigma, a transferência de controle entre os níveis base e reflexivo pode ser feita de forma natural através da interceptação de eventos. No caso particular de Xchart, o sistema de execução sinaliza eventos sempre que a configuração de uma instância é alterada. Através da interceptação desses eventos, é possível monitorar, de forma simples e completa, o comportamento de toda a especificação.

Em nossa arquitetura, existem duas possibilidades para a utilização do nível reflexivo. No primeiro caso, o nível reflexivo é utilizado simplesmente para estender o nível base (operações de remoção de elementos do nível base não são previstas). Esse perfil é utilizado para promover a separação dos domínios que fazem parte da especificação em caráter permanente. A segunda possibilidade é utilizar o nível reflexivo não apenas para estender, mas também para remover elementos do nível base. Nesse caso, o nível reflexivo é

utilizado para intervir no nível base em caráter temporário. Por causa da independência entre os níveis, é possível desfazer/refazer as alterações várias vezes, simplesmente habilitando/desabilitando o nível reflexivo. Em ambas as possibilidades, o código descrito no nível reflexivo poderia ser descrito diretamente no nível base, utilizando apenas os recursos disponíveis na linguagem Xchart. A vantagem em utilizar o nível reflexivo é conceitual e diz respeito à organização da especificação.

A arquitetura reflexiva proposta procurou preservar ao máximo as características da linguagem Xchart. Foram criadas apenas de três novas estruturas para essa linguagem: (1) os diagramas reflexivo; (2) o elemento de ação *reuse*; (3) o elemento de ação *remove*. Essas três estruturas são simples e podem ser tratadas exclusivamente em tempo de compilação. Assim, apenas o compilador TeXchart precisa ser adaptado para tratá-las. Apesar de contrariar o modelo reflexivo, os elementos de ação *reuse* e *remove* foram adicionados (explicitamente) à linguagem Xchart para (1) facilitar a compreensão das operações associadas a esses elementos e (2) para facilitar uma implementação inicial (em tempo de execução). A medida que nossa arquitetura for estendida e passar a ser tratada em tempo de execução, esses elementos de ação deixarão de existir explicitamente na linguagem e as operação associadas passarão a ser realizadas de forma transparente pelo sistema de execução (SE).

Os diagramas reflexivos são utilizados para promover a organização da especificação em diferentes níveis. Os diagramas reflexivos são uma especialização dos diagramas Xchart. Um diagrama reflexivo deve estar associado a pelo menos um diagrama Xchart, ou então, a outro diagrama reflexivo. O nível base (correspondente ao domínio do sistema) é sempre modelado utilizando-se diagramas Xchart. Os demais domínios (ou adaptações do nível base) são descritos em diagramas reflexivos. A arquitetura proposta é parcialmente reflexiva: um diagrama Xchart pode ter ou não um diagrama reflexivo associado a si. Os diagramas reflexivos devem repetir os estados dos diagramas Xchart que serão estendidos, bem como todos os seus estados ancestrais. As regras e transições do diagrama Xchart não precisam ser repetidas no diagrama reflexivo, a não ser no caso em que esses elementos também sejam adaptados no nível reflexivo. Um diagrama reflexivo pode estar associado a múltiplos diagramas Xchart assim como um diagrama Xchart pode ter múltiplos diagramas reflexivos associados a si. A associação de múltiplos diagramas reflexivos os mesmo diagrama Xchart forma uma torre reflexiva. A mesclagem de uma torre reflexiva ocorre de forma *bottom-up*. Do ponto de vista da especificação, é desejável que haja correspondência entre os domínios e os níveis utilizados para descrevê-los nas diversas torres reflexivas.

A separação de domínios em diferentes diagramas existe apenas no nível da especificação. Durante o processo de compilação, o código dos diagramas reflexivos é mesclado ao código dos diagramas Xchart aos quais estão associados. O sistema de

execução não precisa ser alterado porque o código executável é o mesmo da abordagem convencional (sem arquitetura reflexiva). Em tempo de execução, não há *overhead* associado ao mecanismo de reflexão adotado. A responsabilidade pelo código descrito nos níveis reflexivos, assim como acontece quando os domínios estão todos misturados nos diagramas Xchart (abordagem convencional), é do programador. O processo de mesclagem detecta apenas erros relativos à referência (em diagramas reflexivos) de elementos inexistentes nos respectivos diagramas Xchart. Qualquer elemento descrito no nível reflexivo e que não faça referencia a elementos do nível base é considerado uma extensão a esse nível.

Para facilitar a adição de elementos de ação a uma regra já existente do nível base, foi criado o elemento de ação *reuse*. *Reuse* é utilizado para referenciar a ação de uma regra do nível base, evitando repeti-la no nível reflexivo. Durante a mesclagem dos diagramas, *reuse* é substituído pela ação da regra referenciada. Caso uma regra do nível base seja referenciada no nível reflexivo sem a utilização de *reuse*, a ação da regra do nível base é substituída pela ação descrita no nível reflexivo.

O elemento de ação *remove* foi criado para permitir a exclusão de elementos do nível base. Para remover uma regra, basta referenciá-la no diagrama reflexivo substituindo a ação original pelo elemento de ação *remove*. Durante a mesclagem dos diagramas, a presença de *remove* no nível reflexivo causa a eliminação da regra correspondente no nível base. A remoção de regras pode ser utilizada, indiretamente, para remover transições e estados. Ao remover a regra que rotula uma transição, um dos elementos que caracterizam essa transição deixa de existir. Essa situação é utilizada, durante a mesclagem dos diagramas, para indicar a remoção da transição cuja regra foi eliminada. A remoção de todas as transições que ativam um estado implica em que esse estado jamais será ativado e, por causa disso, o estado em questão é eliminado durante a compilação. A remoção de uma transição ou de um estado pode provocar, implicitamente, a exclusão de outras transições e estados (efeito cascata).

A implementação da arquitetura reflexiva implica em alterações em dois componentes do ambiente Xchart: o editor de Xchart e o compilador TeXchart (para realizar a mesclagem dos diagramas). O editor de Xchart funciona como um complemento à nossa arquitetura. A organização da especificação em níveis possibilita oferecer ao usuário diferentes "visões" da especificação e avançados recursos de navegação. A mesclagem dos diagramas é realizada durante a compilação da especificação. O diagrama Xchart é compilado normalmente. Caso haja uma torre reflexiva associada a esse diagrama, a mesclagem dos diagramas é realizada iterativamente, de forma *bottom-up*. Em cada etapa, as extensões descritas no nível reflexivo são realizadas primeiro, na seguinte ordem: primeiro são adicionados os novos estados, em seguida as novas transições e finalmente as novas regras.

Os erros detectados pela mesclagem acontecem quando regras descritas no diagrama reflexivo utilizam os elementos de ação *reuse* ou *remove* e não existem as regras correspondentes no diagrama Xchart. Terminadas as extensões, passam a ser tratadas as operações de exclusão. Após serem realizadas todas as exclusões que estão explícitas no diagrama reflexivo, o efeito em cascata dessas exclusões é tratado. Após terminada a mesclagem com o primeiro diagrama reflexivo, o processo repete-se com os demais diagramas da torre reflexiva.

O impacto das extensões discutidas a seguir na implementação, assim como a impossibilidade de realizar testes que comprovem a eficácia de nossa arquitetura sem utilizar um editor de Xchart, nos levaram a escolher, a princípio, uma abordagem mais simples para a implementação. Nessa abordagem, a mesclagem das torres reflexivas acontece antes da compilação da especificação, utilizando a descrição textual dos diagramas (em TeXchart). Essa implementação é temporária e tem por objetivo, simplesmente, viabilizar a utilização de nossa arquitetura.

5.2 Trabalhos futuros

5.2.1 Metodologia de desenvolvimento

A criação de uma metodologia de desenvolvimento é de grande importância para auxiliar o usuário na utilização da arquitetura reflexiva proposta. Essa metodologia deve guiar o usuário na identificação dos diversos domínios que compõem o gerenciador de diálogo a ser implementado. Uma vez identificados esses domínios, deve ser escolhida a melhor organização para a torre reflexiva, determinando-se quais os domínio serão descritos em níveis distintos, qual a precedência entre os níveis que serão utilizados para descrever esses domínios, etc.

5.2.2 Habilitação/desabilitação em tempo de execução

A arquitetura reflexiva, da forma como foi proposta, prevê a habilitação/desabilitação de níveis reflexivos (domínios) em tempo de compilação. Dessa forma, caso o usuário deseje habilitar/desabilitar um nível reflexivo, toda a especificação precisa ser recompilada. Uma extensão imediata de nossa arquitetura é permitir a habilitação/desabilitação de níveis reflexivos em tempo de execução. Para oferecer suporte a essas operações, o sistema de execução de Xchart precisa ser adaptado. O processo de compilação também precisaria ser revisto. A mesclagem dos diagramas tal como foi proposta não poderia ser realizada. Caso

o SE recebesse o código dos diagramas de uma torre reflexiva mesclado, não seria possível identificar os elementos que pertencem a cada nível. Um problema a ser considerado é a eliminação de níveis intermediários da torre. Na proposta atual, a desabilitação de um nível cujos elementos são referenciados por níveis superiores ocasionaria um erro durante a compilação. A desabilitação desse tipo de nível em tempo de execução deve ser tratada com cuidado.

5.2.3 Metainterface introspectiva

Embora especificações em Xchart tenham a capacidade de invocar atividades, essas atividades interagem com SE de forma bastante restrita (através da IPX). Essas restrições limitam o potencial reflexivo dessas atividades. Caso as atividades pudessem acessar (pelo menos para consulta) a configuração de uma instância, elas poderiam ser utilizadas para fins reflexivos em relação ao gerenciador de diálogo, ou seja, poderiam existir atividades que pertencessem ao próprio subsistema reativo e cuja finalidade seria participar na execução desse subsistema. Atividades reflexivas podem ser úteis, por exemplo, para depurar uma especificação. Imagine que o domínio reflexivo associado a uma especificação intercepta a ocorrência de um determinado evento e grava, em um arquivo, a configuração da instância sempre que o evento é interceptado. Xchart não oferece recursos para manipular arquivos, por isso, deve ser escrita uma atividade com esse objetivo. Essa atividade precisa ter acesso à configuração e à estrutura da instância (valor de variáveis locais, estados ativos, regras habilitadas, etc) para poder gravar os valores requeridos no arquivo. Esse tipo de acesso pode ser conseguido, de forma controlada, através de uma metainterface introspectiva, ortogonal a IPX. Atividades reflexivas utilizam-se da metainterface e da IPX para atingir seus objetivos. As atividades dos demais subsistemas continuam utilizando exclusivamente a IPX.

5.2.4 Sinalização de eventos para diagramas reflexivos

Por ser tratada exclusivamente em tempo de compilação, a arquitetura reflexiva proposta não prevê a sinalização de eventos para diagramas reflexivos. Todos os eventos são sinalizados para os diagramas Xchart. Após a compilação, os diagramas reflexivos são mesclados aos diagramas Xchart, dessa forma, as regras descritas nos diagramas reflexivos efetivamente interceptam e tratam eventos, já que esses são sinalizados apenas em tempo de execução. A partir do instante que os diagramas reflexivos passam a ser conhecidos em tempo de execução, passa a existir a possibilidade de sinalizar eventos diretamente para esses diagramas, aumentando o potencial de nossa arquitetura.

Considere, por exemplo, o caso em que um diagrama reflexivo está associado a múltiplos diagramas Xchart. A possibilidade de sinalizar eventos diretamente para esse diagrama reflexivo evitaria que o mesmo evento fosse sinalizado para todos os diagramas Xchart. Nesse caso, o diagrama reflexivo serviria como uma espécie de "gerente" do conjunto de diagramas Xchart ao qual está associado.

Referências bibliográficas

[1] ____. *JAVA Core Reflection: API and Specification.* Java Soft, Mountain View, CA, USA. Oct 1996.

[2] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts. 1983.

[3] A. Morse and G. Reynolds. Overcome Current Growth Limits in User Interface Development. Communications of the ACM, 36(4):73-81. Apr 1993.

[4] B. C. Smith. *Reflection and Semantics in a Procedural Language.* Ph.D. Thesis, MIT LCS TR-272. Jan. 1982.

[5] B. Gowing and V. Cahill. *Meta-object protocols for C++: The Iguana approach.* Proc. of Reflection '96, pp. 137-152, San Francisco, USA. Apr 1996.

[6] B. Stroustrup. *Run-Time Type Identification for C++.* Proc. of the USENIX C++ Conference, Or. Aug 1992.

[7] B.A. Myers. UIMS's, Toolkits, Interface Builders. Handbook of User Interface Design, Jacob Nielsen, editor. 1997.

[8] C. Maeda, A. Lee, G. Murphy and G. Kiczales. *Open Implementation Analysis and Design.* Tech. Report, http://www.parc.xerox.com/. 1997.

[9] D. Friedman and M. Wand. *Reification: Reflection without meta-physics.* Communications of the ACM, Vol 8. 1984.

[10] D. Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8(3):231-274 Jun 1987.

[11] D.R. Olsen. Presentational, Syntactic and Semantic Components of Interactive Dialogue Specifications. In Günther E. Pfaff, editor, User Interface Management Systems, pp. 125-133. Springer-Verlag. 1985.

[12] F.N. Lucena and H.K.E. Liesenberg. Interface de Programação de Xchart. Em preparação, http://www.dcc.unicamp.br/proj-xchart/texchart, 1998.

[13] F.N. Lucena and H.K.E. Liesenberg. Reflections on Using Statecharts to Capture User Interface Behaviour. Proc. of the XIV Int. Conf. of the Chilean CSS. Oct 1994.

[14] F.N. Lucena, C.N. Júnior, T.H. Yunes, H.K.E. Liesenberg and L.E. Buzato. Especificação da Linguagem TeXchart. Em preparação, http://www.dcc.unicamp.br/projects/Xchart/texchart, 1998.

[15] F.N. Lucena, H.K.E. Liesenberg and L.E. Buzato. Especificação da Linguagem Xchart. Em preparação, http://www.dcc.unicamp.br/proj-xchart/publicacoes.html. 1998.

[16] F.N. Lucena, H.K.E. Liesenberg and L.E. Buzato. Tecnologia Xchart. Em preparação, http://www.dcc.unicamp.br/proj-xchart/publicacoes.html, 1998.

[17] F.N. Lucena, L.E. Buzato and H.K.E. Liesenberg. Xchart: Um Sistema de Gerenciamento de Interfaces Homem-Computador. Workshop de Sistemas Hipermídia, (Wosh'96/SBRC'96), Fortaleza/CE. Maio 1996.

[18] F.N. Lucena, M.M. Harada and H.K.E. Liesenberg. Operational Semantics of Extended Statecharts (XCHARTS). 3rd Workshop on Logic, Language, Information and Computation (WoLLIC'96), Salvador/BA. May 1996.

[19] F.N. Lucena,, C.N. Júnior and H.K.E. Liesenberg. Biblioteca TeXchart. Em preparação, http://www.dcc.unicamp.br/proj-xchart/texchart, 1998.

[20] F.N. Lucena. *Xchart: Um Modelo de Especificação e Implementação de Gerenciadores de Diálogo.* Tese de Doutorado, DCC/IMECC/UNICAMP, Campinas/SP. Dez 1997.

[21] G. Kiczales and A. Paepcke. *Open Implementations and Metaobject Protocols.* Tech. Report, http://www.parc.xerox.com/. 1996

[22] G. Kiczales, J. des Rivières and D. G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press. 1991.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C.V. Lopes, C. Maeda, J. Loingtier and J. Irwin. *Aspect-Oriented Programming.* Proc. of European Conference on Object-Oriented Programming (ECOOP'97), Springer-Verlag Lecture Notes in Computer Science, Finland. Jun 1997.

[24] G. Kiczales, J. Lamping, C.V. Lopes, A. Mendhekar and G. Murphy. *Open Implementations Design Guidelines.* Proc. of the 19th International Conference on Software Engineering. 1996.

[25] G. Kiczales. *Beyond the Black Box: Open Implementation.* IEEE Software, 13(1): 8,10-11. 1996.

[26] G. Kiczales. *Towards a New Model of Abstraction in Software Engineering.* Proc. of International Workshop on Object Oriented in Operating Systems, CA, USA. 1991.

[27] H. Okamura, Y. Ishikawa and M. Tokoro. *AL-1/D: A distributed programming system with multi-model reflection framework.* In IMSA'92 International Workshop on Reflection and Meta-level Architecture. Nov 1992.

[28] H. Okamura, Y. Ishikawa and M. Tokoro. *Metalevel Decomposition in AL-1/D.* In 1st International Symposium on Object Technologies for Advanced Software (ISOTAS'93). Nov 1993.

[29] J. Ferber. *Computational Reflection in Class Based Object-Oriented Languages.* Proc. of the OOPSLA'89, pp. 317-326, LA USA. Oct 1989.

[30] J. Kleinöder and M. Golm. *Metajava: An Efficient Run-Time Meta Architecture for Java.* Tech. Report TR-14-96-03, Universität Erlangen-Nürnberg: IMMD IV. Jun 1996.

[31] J. Laird, P. Rosenbloom and A. Newell. *Chunking in SOAR: The Anatomy of a General Learning Machanism.* Machine Intelligence, Vol 1, No 1. Kluwer Academic Publishers. 1986.

[32] J. McAffer. *Meta-level Programming with CodA.* In ECOOP'95, pp. 190-214. Aug 1995.

[33] J.M. Sobel and D.P.Friedman. *An Introduction to Reflection-Oriented Programming.* Presented at Reflection'96, San Francisco. 1996.

[34] K. Bowen. *Meta-level Techniques in Logic Programming.* Proc. of the International Conference on Artificial Intelligence and its Applications, Singapore. 1986.

[35] M. Golm and J. Kleinöder. *MetaJava - A Platform for Adaptable Operating-System Mechanisms.* Tech. Report TR-14-97-10, Universität Erlangen-Nürnberg: IMMD IV. Apr 1997.

[36] M. Katz and D. Weise. *Towards a new Perspective on Partial Evaluation.* In ACM SIGPLAN, Workshop on Partial Evaluation and Semantics-Directed Program Manipulation. CA, USA. 1991.

[37] M.H. Ibrahim and F.A. Cummins. *KSL: A Reflective Object-Oriented Programming Language*. Proc. of the IEEE Computer Society International Conference on Computer Languages. 1988.

[38] O.S. Júnior. Smart: Um Editor Gráfico para os Diagramas Xchart. Master's thesis, DCC/IMECC/UNICAMP, Campinas/SP. Feb 1996.

[39] P. Cointe. *Metaclasses are First Class: The ObjVlisp Model.* Proc of the OOPSLA'87, Sigplan Notices 22(12), ACM. Dec 1987.

[40] P. Maes. *Concepts and Experiments in Computational Reflection.* OOPSLA 87, ACM Sigplan Notices, 22(12):147-155. Dec 1987.

[41] P. Maes. *Issues in Reflection.* In P. Maes and D. Nardi, editors, Meta-Level Architectures and Reflections. North Holland. 1988.

[42] R. Davis. *Knowledge-Based Systems in Artificial Intelligence.* Davis R. and Lenat D., McGraw-Hill, New York. 1982.

[43] R. Rao. *Implementational Reflection in Silica.* Proc. of ECOOP'91, P. America, ed., vol 512 of Lecture Notes in Computer Science, Springer-Verlag. 1991.

[44] R. Weyrauch. *Prolegomena to a Theory of Mechanized Formal Reasoning.* Artificial Inteligence, Vol 13, No 1,2, Amsterdam, The Netherlands. 1980.

[45] S. Chiba and T. Matsuda. *Design an Extensible Distributed Language with a Meta-Level Architecture.* Proc. of the 7th European Conference on Object-Oriented Programming. 1993.

[46] S. Chiba. *A Metaobject Protocol for C++.* In OOPSLA'95, (30):285-299. Oct 1995.

[47] S. Jefferson and D.P. Friedman. *A Simple Reflective Interpreter.* Lisp and Symbolic Computation, 9(2/3):181-202. May/Jun 1996.

[48] S. Matsuoka, T. Watanabe, Y. Ichisugi and A. Yonezawa. *Object-Oriented Concurrent Reflective Architectures.* In ECOOP'91. Jul 1991.

[49] T. Watanabe and A. Yozenawa. *Reflection in an Object-Oriented Concurrent Language.* Proc. of the OOPSLA'88 Conference, Sigplan Notices 23(11). ACM. Nov 1988.

[50] Y. Yokote, F. Teraoka, and M. Tokoro. *A reflective architecture for an object-oriented distributed operating system.* In ECOOP'89. 1989.

[51] Y. Yokote. *The Apertos Reflective Operating System: The Concept and Its Implementation.* Proc. of the OOPSLA'92 Conference, pp. 414-434, ACM. Oct 1992.