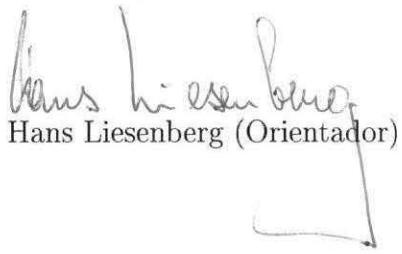
Sins: Um Editor Xchart na forma de Plugin para o Ambiente Eclipse

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Diogo Kollross e aprovada pela Banca Examinadora.

Campinas, 10 de outubro de 2007.



Hans Liesenberg (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Kollross, Diogo
K834e Sins: um editor Xchart na forma de plugin para o ambiente elipse / Diogo Kollross-- Campinas, [S.P. :s.n.], 2007.

Orientador : Hans Kurt Edmund Liesenberg
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Linguagem de programação (Computadores). 2. Software - Desenvolvimento. 3. Sistemas operacionais distribuídos (Computadores). I. Liesenberg, Hans Kurt Edmund. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Sins: An Xchart editor as a plugin for the eclipse environment.

Palavras-chave em inglês (Keywords): 1. Programming languages (Electronic computers). 2. Software – Development. 3. Distributed operating systems (Computers).

Área de concentração: Sistemas de Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Hans Kurt Edmund Liesenberg (IC-UNICAMP)
Prof. Dr. Fábio Nogueira de Lucena (UFG)
Prof. Dr. Luiz Eduardo Buzato (IC-UNICAMP)

Data da defesa: 10/10/2007

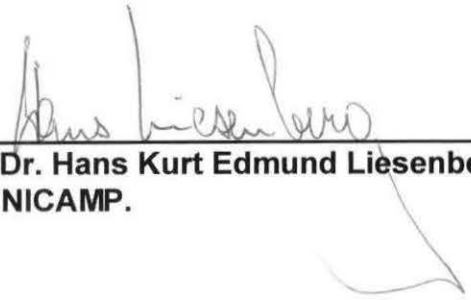
Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 10 de outubro de 2007, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Fábio Nogueira de Lucena
Universidade Federal de Goiás.

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP.



Prof. Dr. Hans Kurt Edmund Liesenberg
IC - UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

Sins: Um Editor Xchart na forma de Plugin para o Ambiente Eclipse

Diogo Kollross

Outubro de 2007

Banca Examinadora:

- Hans Liesenberg (Orientador)
- Fábio Nogueira de Lucena
- Luiz Eduardo Buzato
- Ariadne Maria Brito Rizzoni Carvalho (Suplente)

Resumo

Sistemas reativos têm grande importância em muitas áreas da engenharia e da computação, mas a qualidade e maturidade das metodologias e ferramentas de apoio ao desenvolvimento deixam a desejar em relação às voltadas a sistemas transformacionais. Uma das metodologias de destaque é a Arquitetura Orientada a Modelos, onde os sistemas reativos são descritos por modelos que podem ser diretamente traduzidos em formas executáveis. A linguagem mais bem sucedida na modelagem de sistemas reativos é Statechart, que deu origem a variações como os diagramas de máquinas de estado do padrão UML e à linguagem Xchart. Essa linguagem é uma extensão de Statechart que introduz construções para controle de processos externos, história de ativações e hierarquização de eventos. Para superar as limitações da ferramenta já existente para edição de diagramas Xchart conhecida como Smart, foi desenvolvido o editor Sins (Sins Is Not Smart), implementado como plugin para o ambiente integrado de desenvolvimento Eclipse. Com o editor Sins é possível editar os diagramas através de manipulação direta, diagramar a especificação automaticamente e gerar o código fonte correspondente na linguagem textual TEXchart. O algoritmo de layout implementado é uma variação do algoritmo de Sugiyama, modificado para melhorar a legibilidade do diagrama ao garantir a consistência na apresentação de suas estruturas e gerar mapas semelhantes aos desenhados livremente.

Abstract

Reactive systems have great importance in many areas of Engineering and Computing, but the quality and maturity of the development support methodologies and tools lack when compared to those directed to transformational systems. One of the outstanding methodologies is Model Oriented Architecture, where the reactive systems are described by models that can be directly translated to executable form. The best succeeded language for modeling of reactive systems is Statechart, which is the origin of variations like state machine diagrams from the UML standard and the Xchart language. This language is an extension of Statechart that introduces elements for external process control, activation history and hierarchization of events. To overcome the limitations of the already existing tool for the edition of Xchart diagramas known as Smart, the Sins editor was developed (Sins Is Not Smart), implemented as a plugin for the Eclipse IDE. With the Sins editor it is possible to edit diagramas through direct manipulation, layout the specification automatically and generate the corresponding source code in the textual language TEXchart. The implemented layout algorithm is a variation of the Sugiyama algorithm, modified for better legibility of the diagram by assuring consistency in the presentation of its structures and generation of layouts similar to those freely drawn.

Agradecimentos

Agradeço, antes de qualquer coisa, a meus pais por todo o apoio que me deram não só durante o mestrado mas durante toda os momentos, bons e ruins, pelos quais já passei.

Agradeço também às minhas irmãs, Nicole e Bianca, que sempre estiveram ao meu lado como amigas, confidentes e conselheiras.

Agradeço ao meu orientador Hans Liesenberg, pela paciência e compreensão com minhas dificuldades e pela orientação em todos os momentos.

Agradeço aos professores e funcionários do Instituto de Computação por me receberem, me ensinarem e me ajudarem com todos os problemas que surgiram nesses dois anos de convivência.

Agradeço aos amigos, novos e antigos, próximos e distantes, por me ajudarem a manter a sanidade mental durante todo esse tempo, por me ensinarem sobre a vida e por dividirem as risadas e os lamentos.

Agradeço, finalmente, à minha namorada Amanda por me dar forças em todas as horas, por dar sentido à minha vida e por me fazer acreditar que amar vale a pena.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Sistemas interativos têm grande importância na engenharia e na implementação de sistemas computacionais complexos. Uma das ferramentas mais importantes para o projeto, compreensão e depuração de sistemas reativos é a utilização de modelos. Esses modelos, manipulados na forma de diagramas, podem servir de entrada em processos que os simulam ou geram código executável equivalente, diminuindo a distância entre o projeto e a implementação de um sistema.

Dentre as linguagens existentes para a especificação de modelos de sistemas reativos, uma das linguagens mais bem sucedidas é Statechart [1], servindo de base para os diagramas de máquinas de estado do padrão UML [2]. Criada em meados dos anos 80, Statechart estende a notação de diagramas de transição de estados para facilitar a descrição de sistemas reativos complexos.

Observando-se as limitações da linguagem Statechart, foi criada a linguagem Xchart, estendendo as construções disponíveis na primeira. Xchart é também um ambiente de desenvolvimento e execução, oferecendo ferramentas para o projeto, depuração e execução distribuída de sistemas reativos especificados em Xchart.

Atualmente, é possível usar a tecnologia Xchart para a especificação de sistemas reativos de duas formas: usando a linguagem TEXchart, equivalente textual da linguagem Xchart, ou usando o editor Smart, que oferece a visualização de diagramas Xchart mas limita sua edição à manipulação indireta, através da árvore de inclusão de estados e de comandos de menu. Além dessa limitação, o editor Smart está disponível apenas para Windows, o que não acompanha a migração corrente do ambiente Xchart para a plataforma Java.

A solução descrita nesta dissertação é a criação de um editor de diagramas Xchart que supera as limitações das opções atuais: manipulação visual e direta dos diagramas e adoção da plataforma Java, estendendo o ambiente integrado de desenvolvimento Eclipse. No próximo capítulo são explicados os conceitos envolvidos no problema tratado e os

trabalhos já existentes na área. No capítulo 3, são descritos em detalhes a tecnologia Xchart, a linguagem visual Xchart e a linguagem textual TEXchart. No capítulo 4 o editor Sins é apresentado como solução para o problema de especificar sistemas reativos, detalhando as funcionalidades oferecidas. No capítulo 5 é descrita a implementação do editor, o algoritmo de layout automático de diagramas proposto e os resultados obtidos. No último capítulo são analisadas as contribuições deste trabalho e direções são sugeridas para desenvolvimentos futuros.

Capítulo 2

Contextualização e Terminologia

2.1 Introdução

Neste capítulo serão apresentados e discutidos tópicos relevantes ao problema de desenvolver sistemas reativos e a criação de um editor de diagramas Xchart. Na seção 2.2 serão definidos os conceitos de sistemas reativos e as técnicas existentes para seu desenvolvimento. Esse desenvolvimento pode ser facilitado com a utilização de arquiteturas orientadas a modelos, como descrito na seção 2.3. Tais modelos são representados por notações visuais na forma de diagramas, que podem ser editados com o auxílio de editores, discutidos na seção 2.4. A utilização desses editores se beneficia da funcionalidade de layout automático, comentada na seção 2.5. Outros conceitos, referenciados ao longo deste documento, são apresentados na última seção deste capítulo.

2.2 Sistemas reativos

2.2.1 Definições

Sistema reativo é definido por Benveniste como "...um sistema que mantém uma interação permanente com seu ambiente" [3], isto é, um sistema que reage a eventos gerados externa ou internamente, respondendo com ações que podem afetar o seu estado interno ou o seu ambiente. De forma similar, sistemas de tempo real podem ser definidos como sistemas reativos cujas respostas devem obedecer a limites temporais definidos externamente. Quando a quebra desses limites implica na falha do sistema, se diz que o sistema é de tempo real *hard* (*hard real-time*); quando a quebra desses limites não leva à falha do sistema, o sistema é dito de tempo real *soft* (*soft real-time*).

Um sistema transformacional, por sua vez, recebe uma entrada, a processa e gera uma saída correspondente, finalizando assim sua execução. Enquanto sistemas reativos

geram sua saída à medida que aceitam partes de sua entrada, que podem ser recebidas em momentos arbitrários, sistemas transformacionais têm sua entrada completamente disponível no início de sua execução, gerando sua saída completamente em um momento posterior.

Sistemas reativos são aplicados em muitas áreas importantes da indústria eletrônica e de informática, como protocolos de comunicação, processamento de sinais, interfaces de usuário e sistemas interativos em geral, controle de processos industriais e sistemas complexos de controle e monitoramento.

Apesar de sua importância, as técnicas de desenvolvimento de sistemas reativos não são tão estudadas e, conseqüentemente, tão bem conhecidas, quanto aquelas para sistemas transformacionais. Isso se reflete na relativa falta de trabalhos de pesquisa, linguagens e ferramentas de projeto e programação de sistemas reativos. Recentemente, essa diferença no tratamento a ambos os tipos de sistemas, aliada à crescente necessidade de implementação de sistemas reativos complexos e sem erros, levou à uma maior quantidade de pesquisa nessa área.

Isso resultou na proposta de um número de técnicas, linguagens e ferramentas que buscam facilitar o desenvolvimento desses sistemas. Essas técnicas podem ser agrupadas de acordo com a abstração usada para a modelagem, pelo desenvolvedor, para descrever e raciocinar sobre o sistema a ser desenvolvido. Os principais grupos de técnicas são listados na seção 2.2.3 e incluem a conexão de aplicações clássicas, gramáticas, redes de Petri e de transição e modelo de eventos.

Uma categoria de sistema relacionada à dos sistemas reativos é a dos sistemas concorrentes. Sistema concorrente, segundo Roscoe [4], é aquele "...onde existe mais de um processo a cada momento". Esses processos precisam interagir entre si para atingir os propósitos do sistema, levando à necessidade de comunicação entre os componentes do sistema. Sistemas concorrentes se diferenciam dos tradicionais sistemas seqüenciais porque nestes, em oposição aos concorrentes, existe apenas um processo em execução a cada momento.

É comum que sistemas reativos possam, pelas características do problema a ser resolvido, ser modelados de forma modular, com a composição de subsistemas que se comunicam, tal qual um sistema concorrente. Assim, muitos sistemas reativos são modelados como sistemas concorrentes, aproveitando-se as técnicas específicas para o desenvolvimento desses sistemas.

Da forma similar aos sistemas reativos, o desenvolvimento de sistemas concorrentes é uma área relativamente nova de pesquisa, com o surgimento recente de novas técnicas, ferramentas e linguagens. Entre essas técnicas se destacam também as redes de Petri, a lógica temporal, os cálculos de processos e, mais recentemente, o modelo de ator. Além das redes de Petri, que também são utilizadas na descrição de sistemas reativos, os cálcu-

los de processos resultaram em um número de linguagens de programação adaptadas ao desenvolvimento de sistemas concorrentes e reativos, acompanhadas de ferramentas poderosas para a depuração e a verificação formal desses sistemas. Algumas dessas linguagens são descritas na seção 2.2.3.5.

2.2.2 Sincronicidade

Sistemas reativos podem ser classificados quanto ao seu comportamento: síncronos ou assíncronos. De acordo com Benveniste [3], um sistema síncrono têm três características que o definem: a saída é síncrona com a entrada, as ações internas são instantâneas e a comunicação é feita através de *broadcast* global.

A primeira característica é a mais importante para essa definição. A entrada de um sistema reativo é a recepção de eventos ou sinais externos ou internos; a saída é a resposta do sistema, através da execução de ações, geração de novos eventos e muitas vezes acompanhada pela mudança de estado do sistema. Em sistemas síncronos, a saída é intimamente ligada à entrada: o tratamento de um evento só inicia após o término da resposta a todos os eventos anteriores.

A execução instantânea das ações internas se relaciona com a sicronicidade entre entradas e saídas, podendo ser vista como conseqüência desta. Num sistema síncrono, assume-se que as ações internas, isto é, recepção de e resposta a eventos, são instantâneas. Apesar desse conceito não se verificar em implementações reais, onde sempre existe um retardo entre a recepção de um evento e a sua respectiva reação, esse período de tempo é abstraído para facilitar a modelagem do comportamento do sistema. Essa simplificação torna possível a composição, compreensão e análise, inclusive automatizada e formal, de sistemas reativos complexos.

Finalmente, a comunicação entre os diversos componentes do sistema reativo ocorrem através de broadcast: cada evento gerado, interna ou externamente, é recebido em qualquer parte do sistema capaz de tratá-lo e isso ocorre no mesmo instante em que a geração do evento ocorre. Essa característica está relacionada às duas primeiras, mantendo a coerência lógica desta definição de sistema reativo síncrono e possibilitando o raciocínio formal sobre ele.

2.2.3 Técnicas de desenvolvimento

2.2.3.1 Conexão de aplicações clássicas

Uma das primeiras e mais simples técnicas é a conexão de aplicações clássicas. Essa conexão realiza-se através de primitivas de comunicação inter-processos do sistema operacional. Nessa técnica, o sistema é subdividido em subsistemas que são implementados

através de técnicas tradicionais de desenvolvimento e se conectam por recursos como envio de mensagens através de camadas de rede e compartilhamento de recursos por meio de canais ou memória compartilhada.

Atualmente, grande parte dos sistemas reativos é implementada através dessa técnica. Entre as desvantagens dessa técnica estão as dificuldades na depuração e manutenção, assim como a impossibilidade de verificação formal das propriedades do sistema reativo desenvolvido. Esses problemas advêm da falta de garantias de desempenho e determinismo por parte das primitivas de comunicação interprocessos dos sistemas operacionais, do pouco ou nenhum embasamento em fundações matemáticas ou formais e no pouco controle existente sobre a implementação de cada um dos componentes do sistema reativo.

2.2.3.2 Gramáticas

O uso de gramáticas para descrever o comportamento de sistemas reativos é outra opção entre as técnicas possíveis [5]. Nesse caso, assume-se que as entradas e correspondentes saídas válidas compõem uma linguagem que pode ser descrita por uma gramática. Os eventos externos e as respostas correspondentes são modelados como símbolos da gramática e as regras descrevem o comportamento do sistema. As notações costumam ser variações da BNF (*Backus-Naur Form*), com a utilização de gramáticas livres de contexto.

A grande vantagem dessa técnica é o embasamento formal e lógico oferecido pelas gramáticas, que têm suas propriedades bastante conhecidas. A simplicidade trazida por regras de reescrita faz com que gramáticas sejam usadas como base para definir formalmente linguagens equivalentes em outras notações, como o mapeamento visual baseado em gramáticas proposto por Chau [6].

Por outro lado, o uso de gramáticas traz um número de desvantagens para a modelagem de sistemas reativos. Gramáticas conseguem apenas descrever a sintaxe do diálogo: muitas vezes o sistema precisa tomar decisões baseadas também no estado da aplicação ou a partir de outras informações de ordem semântica. Esse tipo de ligação é dificultada pelas características das gramáticas; os estados e a transição entre eles são implícitos e não existe contexto para as decisões tomadas a cada passo.

Outra desvantagem resulta da utilização prática de gramáticas. Uma gramática descreve uma linguagem que é reconhecida, em sistemas reais, por um *parser*. Usualmente, na modelagem de sistemas reativos, cada regra da gramática é associada a uma ação que corresponde à resposta do sistema a um dado evento. A execução dessa ação depende das propriedades da implementação do parser que reconhece a gramática. Um parser *top-down* executará as ações no momento em que as regras forem casadas com a entrada de eventos. Diferentemente, um parser *bottom-up* as executará apenas no momento da redução. Essa diferença pode ter conseqüências importantes no comportamento do sistema modelado.

Uma terceira desvantagem é a dificuldade de utilizar gramáticas para descrever interações concorrentes. Gramáticas são inerentemente adaptadas para modelar o comportamento de sistemas que respondem a eventos com seqüências bem definidas. A recepção de eventos em qualquer ordem, por exemplo, aumenta o tamanho e a complexidade da gramática correspondente.

Além disso, gramáticas livres de contexto tradicionais não diferenciam entre símbolos correspondentes a eventos recebidos pela aplicação e aqueles relativos à respostas do sistema. Isso exige alguma forma de diferenciar entre esses dois tipos de símbolos. Uma forma de fazer isso é através de anotações nas regras da gramática, como sugerem Shneiderman [7] e Jacob [8].

2.2.3.3 Redes de Petri

Redes de Petri foram um dos primeiros formalismos criados para a descrição do comportamento de sistemas concorrentes. Uma rede de Petri é composta de três tipos de elementos [9]: posições, transições e arcos direcionais que ligam posições a transições e vice-versa, mas não conectam posições ou transições entre si. Associados a um arco está o número de *tokens* a serem consumidos ou gerados com o disparo da transição correspondente. Cada posição pode armazenar um número de tokens que são transferidos não-deterministicamente e atomicamente de uma posição à outra quando as transições são disparadas. Transições podem ser disparadas não-deterministicamente quando estão habilitadas, isto é, todos os arcos de entrada estão ligados a posições que contêm o número exigido de tokens.

A notação usual para redes de Petri é visual, similar à de diagramas de transição de estados. Posições são representadas por circunferências, tokens por círculos desenhados dentro das posições, transições por barras e arcos por linhas ou curvas conectando suas extremidades. Notações alternativas também foram propostas, como *GRAFCET* [10] e diagramas de função seqüencial [11] (*SFC – Sequential Function Chart*).

As propriedades matemáticas de redes de Petri foram bastante estudadas e são bem conhecidas, o que facilita a análise e verificação de sistemas modelados por essa técnica. Além disso, a semântica bem definida permite a execução direta de redes de Petri por sistemas reais.

Por outro lado, essa técnica não se adapta bem a descrição de sistemas complexos. Redes de Petri não oferecem recursos para a descrição modular de sistemas, dificultando sua utilização em aplicações não-triviais, onde existe um número significativo de estados ou posições. Outra dificuldade é a o não-determinismo inerente aos sistemas modelados por essa técnica, que pode trazer problemas na depuração dos sistemas.

2.2.3.4 Redes de transição

Uma rede de transição é um conjunto de estados e transições que relacionam estados, como por exemplo em máquinas de estado finitos. Um evento é associado a cada transição, ativando-a quando ocorre e alterando o estado corrente do sistema. As notações mais comuns para essa técnica são as baseadas em diagramas de transição de estados, em que estados são representados por polígonos ou circunferências e transições por linhas ou curvas.

Essa técnica é adequada para descrição do comportamento de sistemas reativos simples, assim como reconhecedores de linguagens pouco complexas. A possibilidade de modelagem de estados adapta-se bastante ao modelo mental usual de sistemas reativos, em que muitas vezes a resposta do sistema depende do seu estado. A larga experiência no uso de redes de transição, aliada ao seu determinismo e forte fundação formal facilitam a verificação e análise de sistemas descritos por essa técnica.

Uma das desvantagens de redes de transição é o pouco apoio ao projeto modular e à reutilização de especificações já existentes. Outra grande dificuldade é a modelagem de estados concorrentes e as transições entre eles porque isso acarreta na adição de muitos elementos ao diagrama. Essas desvantagens tornam inviável, na prática, a utilização de diagramas de transição de estados para especificar sistemas não-triviais.

Para superar essas desvantagens, Harel propôs uma extensão aos diagramas de transição de estados conhecida como Statechart [1], em que existem construções para simplificar a utilização de estados concorrentes, comunicação entre os componentes da aplicação e a composição modular do sistema. Posteriormente, extensões dessa linguagem como Argos [12], Xchart [13] e SyncChart [14] foram criadas, adicionando construções que aumentam o poder e a facilidade de utilização de redes de transição. Todas essas linguagens contam com ambientes e ferramentas que facilitam o projeto, execução e depuração de sistemas reativos. Mais recentemente, a organização W3C definiu uma notação textual equivalente a um subconjunto de Statechart denominada SCXML [15], baseada na linguagem XML.

2.2.3.5 Modelo de eventos

Nessa técnica um sistema reativo é modelado como um conjunto de tratadores de eventos, que podem ser gerados externa ou internamente. Cada tratador é capaz de modificar o estado do sistema e gerar novos eventos que permitam a comunicação com outros tratadores do mesmo sistema ou com o ambiente externo.

O modelo de eventos está fortemente ligado a formalismos utilizados para descrição de sistemas e processos concorrentes, tais como a lógica temporal e cálculo de processos. Várias linguagens formais foram criadas a partir desses formalismos [16]. As três principais, das quais a maior parte das outras linguagens deriva, são a CSP (*Communica-*

ting sequential processes), CCS (*Calculus of communicating systems*) e ACP (*Algebra of Communicating Processes*). Além dessas, muitas outras linguagens voltadas ao desenvolvimento de sistemas concorrentes foram criadas, como Occam, Esterel, Lustre, SOL [17] e Quartz [18], e podem ser usadas segundo essa técnica. Finalmente, linguagens como Ada oferecem primitivas para descrever concorrência segundo o modelo de eventos.

O uso mais disseminado dessa técnica é relativamente recente e no momento é uma área ativa de pesquisa. A fundamentação de linguagens em formalismos matemáticos facilita a verificação automática de sistemas descritos em tais linguagens. Uma desvantagem do modelo de eventos é a potencial baixa abstração [19], quando linguagens pouco adaptadas a essa técnica sejam utilizadas.

2.2.4 Sistemas interativos

Segundo o modelo de interface Seeheim [20, 21], uma forma de estruturar conceitualmente um sistema interativo, isto é, um sistema computacional com o qual um usuário interage para atingir determinados objetivos, é dividindo-o em uma camada computacional (de aplicação) fracamente acoplada a uma camada interativa (de interface de usuário). A camada computacional é responsável pela funcionalidade necessária para que o usuário possa realizar suas tarefas. A camada interativa intermedia o acesso, pelo usuário, aos recursos oferecidos pela aplicação, correspondendo à interface do sistema. Tanto o sistema interativo como um todo, quanto suas camadas, podem ser vistos como sistemas reativos e, conseqüentemente, se beneficiar das técnicas de desenvolvimento desse tipo de sistema.

As principais atribuições da interface de usuário são:

- identificar as intenções do usuário;
- convertê-las em requisições para a aplicação;
- exibir os resultados produzidos pela camada computacional.

No modelo Seeheim a camada interativa é dividida em três subcamadas, como mostrado na figura 2.1:

- Apresentação (nível léxico da interface de usuário), responsável pelo representação dos elementos de interface necessários à comunicação com o usuário, pela recepção dos sinais enviados pelo usuário e exibição das respostas da aplicação contidas nos elementos de interface;
- Controle de diálogo (nível sintático), responsável "...pela sintaxe de interação com o usuário, pelo layout de objetos de interação e pela comunicação entre a apresentação

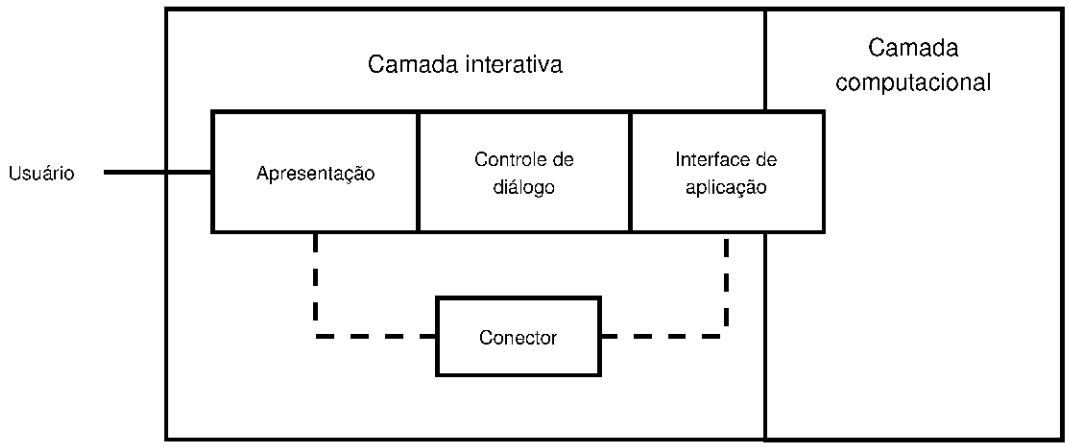


Figura 2.1: Modelo Seeheim

e a aplicação, inclusive conversões entre dados manipulados por esses componentes." [22]; essa camada traduz as entradas do usuário em requisições à aplicação, atualizando o estado dos elementos de interface de acordo com a resposta da camada computacional e convertendo os formatos dos dados trocados quando necessário;

- Interface de software (nível semântico), que interage com a camada computacional por meio de um protocolo de comunicação bem definido, garantindo um desejável fraco acoplamento. Dessa forma, obtém-se independência de diálogo da aplicação: modificações na camada computacional não implicam necessariamente em mudanças na camada interativa e vice-versa.

Além das três subcamadas, é possível incluir na camada interativa um conector para realimentação semântica rápida, introduzido por questões de eficiência. Esse conector liga a subcamada de apresentação à subcamada de interface de software, permitindo, por exemplo, a exibição e modificação de valores de um banco de dados sem a intermediação da subcamada de controle de diálogo.

2.2.4.1 Controle de diálogo

O controle de diálogo é responsável pela ligação entre a camada de apresentação, com a qual o usuário interage diretamente, e a funcionalidade oferecida pela aplicação. Entre suas atribuições estão:

- controle das transições possíveis entre as partes do diálogo que ocorre entre o usuário e a aplicação;

- transformação das entradas do usuário em dados manipuláveis pela aplicação;

- ativação das funcionalidades da aplicação a partir de eventos gerados pelo usuário;
- transformação dos resultados produzidos pela aplicação em signos compreensíveis pelo usuário e
- atualização do estado da camada de apresentação a partir de eventos gerados pela aplicação.

Sendo um sistema reativo, as técnicas de desenvolvimento desse tipo de sistema se aplicam também a componentes de controle de diálogo.

2.3 Arquitetura orientada a modelos

Arquitetura orientada a modelos (*MDA – Model Driven Architecture*) é "...uma forma de organizar e gerenciar arquiteturas empresariais apoiando-se em ferramentas automatizadas e serviços para definir os modelos e facilitar as transformações entre diferentes tipos de modelos" [23]. A MDA surgiu da observação de duas tendências no desenvolvimento de sistemas de negócios nas grandes empresas. A primeira, de que a visão e organização de sistemas como arquiteturas orientadas a serviços (*SOA Service Oriented Architecture*), em que a funcionalidade necessária é organizada em "...uma coleção de serviços encapsulados..." comunicando-se por meio de "...operações definidas através de suas interfaces de serviço". A segunda, de que é possível ver os sistemas de uma empresa como uma linha de produtos, em que padrões comuns de desenvolvimento podem ser reconhecidos, disseminados e aplicados para aumentar a produtividade do desenvolvimento e manutenção dos sistemas, permitindo inclusive a utilização de ferramentas automatizadas para esse fim.

Um modelo, segundo o conceito utilizado nas engenharias, é uma abstração de um sistema real. Na Engenharia de Software, é longa a tradição no uso de modelos, tanto para o projeto de novos sistemas quanto para a descrição de sistemas já existentes. Modelos são úteis para comunicar visões sobre sistemas às respectivas partes interessadas, analisar suas propriedades e prever as implicações de mudanças.

Um mesmo sistema pode dar origem a um número de modelos, onde cada modelo oferece uma visão sobre determinados aspectos ou dimensões do sistema. É comum a transformação de um modelo em outro tipo de visão de mesmo nível de abstração ou entre diferentes níveis de abstração de uma mesma visão. Ambas as transformações podem ser facilitadas ou até mesmo automatizadas com o uso de regras de transformação e informações adicionais sobre o modelo ou sistema.

A MDA assenta-se em quatro princípios segundo a visão do *Object Management Group* (*OMG*):

- modelos devem ser expressos em notações bem definidas para facilitar a compreensão de sistemas complexos;
- a construção de sistemas pode ser organizada em um conjunto de modelos e transformações entre eles;
- descrever modelos com o uso de um conjunto de metamodelos com embasamento formal facilita a integração e transformação automatizadas entre modelos;
- a aceitação de uma filosofia de desenvolvimento centrada em modelos exige o uso de padrões abertos que incentivem a competição entre implementações de ferramentas e serviços.

A partir desses princípios, três tipos principais de modelos são reconhecidos [24]:

- Modelo Independente de Computação (*CIM — Computation Independent Model*), equivalente ao modelo de negócios da aplicação;
- Modelo Independente de Plataforma (*PIM – Platform Independent Model*), que corresponde à implementação independente de plataforma dos processos ou estruturas descritas no CIM;
- Modelo Específico de Plataforma (*PSM – Platform Specific Model*), resultante da transformação de um PIM, através de regras definidas em uma linguagem de transformação de modelos, em uma visão menos abstrata e dependente de plataforma do modelo original.

A definição de plataforma não ocorre de forma absoluta e estática: esse conceito depende fortemente do contexto e necessidades do desenvolvimento, relacionando-se principalmente ao nível de abstração do modelo. Um modelo de um sistema (independente de plataforma) poderia ser transformado em dois modelos alternativos (dependentes de plataforma): um em que a interação com o usuário ocorre via *web* e outro em que ela ocorre através da linha de comando. Esses dois modelos (agora considerados independentes de plataforma) poderiam ser transformados em um número de modelos (dependentes de plataforma, do ponto de vista desta transformação) que utilizariam tecnologias diversas para a execução do sistema, como *Java Server Pages* [25]ou *Active Server Pages* [26]. Percebe-se, assim, que a dependência ou independência de plataforma é um conceito subjetivo e altamente dependente do contexto.

Em resumo, a MDA oferece uma visão do desenvolvimento de sistemas como um processo que envolve a criação, combinação e refinamento de modelos através de transformações sucessivas. Tanto os modelos quanto as transformações são definidos precisamente através de metamodelos descritos em linguagens formais como MOF (*Meta-Object Facility*), OCL (*Object Constraint Language*) e padrões como QVT (*Queries/Views/Transformations*).

2.3.1 Diagramas

Grande parte dos modelos utilizados na MDA podem ser descritos em linguagens visuais. Uma das principais e mais conhecidas linguagens, padronizada pela OMG [27], é a Linguagem Unificada de Modelagem (*UML — Unified Modeling Language*), voltada à modelagem de sistemas segundo o paradigma de projeto Orientado a Objetos. Um modelo UML de um sistema é composto por uma coleção de diagramas que descrevem diferentes visões de suas partes (diagramas estruturais) e comportamento (diagramas comportamentais). Diagramas estruturais incluem diagramas de classes, de componentes e de objetos; diagramas comportamentais incluem diagramas de atividade, de caso de uso, de comunicação, de seqüência e de máquina de estados.

Um diagramas de máquina de estados, como definido em UML, usa uma notação que é uma variação da linguagem Statechart, criada por Harel nos anos 80. Essa linguagem adiciona três novos recursos às tradicionais notações baseadas em diagramas de estados finitos: concorrência, hierarquia e comunicação. Dessa forma, sistemas muito mais complexos podem ser modelados, superando as limitações inerentes às máquinas de estados tradicionais.

Xchart é uma extensão à linguagem Statechart, criada nos anos 90, que adiciona recursos inicialmente voltados à descrição do controle de diálogo de interfaces de usuário, mas que se mostraram úteis para a modelagem de sistemas reativos em geral. Entre esses recursos estão a função de história, o controle de processos externos e a hierarquização de eventos.

2.4 Edição de diagramas

Para permitir o uso de diagramas no contexto da MDA, é necessária a existência de ferramentas computacionais que apóiem a criação, modificação, execução e depuração de modelos. Um dos tipos mais importantes de ferramentas é o editor de diagramas, uma aplicação responsável por facilitar a entrada e visualização de diagramas e a sua ligação com as outras ferramentas que fazem parte de um processo de desenvolvimento orientado a modelos, como simuladores e sistemas de transformação.

2.4.1 Editores de diagramas

Como dito acima, editores de diagramas objetivam facilitar a entrada e simulação de modelos descritos em linguagens visuais (diagramas). Sendo assim, costumam aproveitar os recursos oferecidos por interfaces de usuário gráficas (*GUI*) e pelo seu estilo de interação mais disseminado, *WIMP*. Usualmente, um diagrama é visualizado em sua própria janela e editado através de técnicas de manipulação direta.

As ações aplicáveis ao diagrama são disponibilizadas em menus e botões e podem ser classificadas em três categorias: ações de diagrama (referentes ao diagrama como um todo), ações de elemento (relativas aos elementos que compõem o diagrama) e ações de visão (que modificam as visões possíveis do diagrama). Ações de diagrama incluem criar novo diagrama, abrir diagrama existente, salvar diagrama modificado e imprimir diagrama. Ações de elemento afetam coleções de um ou mais elementos, como por exemplo a criação, movimentação, modificação e remoção de elementos e conexões e o layout automático. Finalmente, ações de visão incluem as mudanças de zoom e a navegação por partes do diagrama.

As ações de elemento, por sua vez, podem apoiar três tipos de edição: livre, orientada à sintaxe e inteligente [28]. Na edição livre, o editor oferece recursos para o desenho de qualquer elemento do diagrama, independente da sintaxe da linguagem visual utilizada, possibilitando a criação de diagramas inválidos ou inconsistentes. Na edição orientada à sintaxe, todas as operações de edição obedecem rigorosamente à sintaxe da notação do diagrama: este nunca está num estado inconsistente e pode-se utilizar operações mais complexas que as oferecidas pela edição livre. Ambas as alternativas têm limitações: a edição livre por permitir a criação de diagramas potencialmente incorretos e a edição orientada à sintaxe por restringir a maneira de trabalho do usuário à operações consistentes mas predefinidas. Assim, um terceiro tipo de edição foi proposto por Chok *et al.*, chamada edição inteligente, em que operações básicas são oferecidas, como na edição livre, mas a consistência do diagrama é garantida através da análise sintática simultânea à edição do diagrama.

Um dos recursos que tem se disseminado largamente entre os editores de diagrama é o layout automático. Este recurso dispõe os elementos de um diagrama de maneira a facilitar a sua leitura e compreensão e libera o usuário do editor para tarefas mais abstratas como a modificação da arquiteturas de um sistema. Como grande parte dos diagramas utiliza uma notação similar à de grafos, em que elementos relacionados se conectam graficamente, muitas das técnicas de layout automático de grafos são aplicadas nos editores de diagramas.

North e Koutsofios [29] desenvolveram o editor de grafos extensível dotty. dotty foi construído como um conjunto de componentes independentes e especializados que podem ser combinados para formar um editor de grafos extensível ou uma interface para um aplicativo que gere informação na forma de grafos. Os componentes envolvidos são um visualizador gráfico programável (lefty), implementações de algoritmos de layout automático (dot e neato) e filtros de grafos (tred, unflatten, gpt e colorize). Diferentemente de outras soluções, dotty utiliza uma linguagem de programação de alto nível customizada, específica para o problema. Essa linguagem permite o controle dos componentes dot e lefty e inclui funcionalidades como definição de funções, arrays escalares e associativas,

espaços de nomes hierárquicos, janelas, menus e entrada e saída de dados. dotty foi estendido para a criação de uma interface para as ferramentas de análise de código fonte cia e cia | (ciao), para a ferramenta de depuração dbx (vdbx) e para a ferramenta de monitoramento de processos distribuídos nDFS (vpm). Uma das desvantagens do dotty é a falta de layout automático incremental de grafos.

Lucas [30] propôs um editor visual para diagramas Statechart em 1993. Esse editor seria uma alternativa ao editor STATEMATE, sistema fechado e proprietário cuja licença de uso básica alcançava dezenas de milhares de dólares. O editor seria orientado a objetos, oferecendo uma palheta de elementos que seriam usados para desenhar o diagrama livremente. Seria possível desativar o desenho de estados contidos em um estado composto para simplificar a visualização do diagrama, assim como ativar um algoritmo de layout automático sempre que desejado pelo usuário. Esse algoritmo de layout reposicionaria e redimensionaria estados e transições do diagrama para atender a três critérios: estados relacionados são desenhados próximos um do outro, o número de cruzamentos entre transições é minimizado e transições não encostam em estados. São propostos vários algoritmos para o layout automático, entre eles a simulação de forças de atração e repulsão entre os estados, a distribuição de estados relacionados (ligados entre si por transições) em círculos concêntricos, empacotamento de subestados concorrentes usando um algoritmo original baseado em bin-packing denominado AFFDH (alternating first-fit, decreasing-height) ou algoritmos de força bruta, como permutações (assumindo que o número de subestados concorrentes é pequeno). A implementação desse editor seria direcionada a sistemas Unix, possivelmente usando como base o editor de grafos EDGE (Extensible Directed Graph Editor), que é escrito em C++ para o ambiente X Window e é extensível quanto ao tipo de grafos que podem ser editados, ao formato de armazenamento dos grafos e aos algoritmos de layout automático disponíveis ao usuário. Os diagramas Statechart seriam armazenados em um arquivo inspirado nas ferramentas lex e yacc, estendendo a linguagem C−+ com construções que descreveriam os elementos de um diagrama e as relações entre eles. Condições e ações associadas a eventos seriam escritas em C−+ em um editor de textos embutido no editor de diagramas e seriam armazenadas no mesmo arquivo do diagrama. Informações sobre posição e tamanho de elementos seria codificada em comentários C | com formatação específica.

Castelló et al. [31, 32] descrevem a ferramenta ViSta de visualização e edição estáticas e interativas de diagramas Statechart. Essa ferramenta permite e criação de diagramas a partir de modelos textuais ou através da edição direta e visual de diagramas. Quando são usados modelos textuais, um assistente de modelo permite a entrada de texto estruturado que descreve o comportamento do sistema reativo. O diagrama Statechart pode ser gerado a partir do modelo textual entrado, desenhado por algoritmos de layout automático. Os algoritmos usados são baseados no algoritmo de Sugiyama [33], com modificações como

o uso de vértices fictícios para manter a posição relativa de vértices e manter o mapa mental do diagrama nas operações de inserção e remoção. Arestas que atravessam limites entre estados compostos são desviadas para vértices Goto contidos no superestado que contém o vértice de origem, rotulados com o nome do estado de destino, para simplificar o algoritmo de layout. Estados compostos são desenhados com orientação vertical ou horizontal, alternadamente, para minimizar a área usada pelo diagrama e aproximar a relação de aspecto de 1: estados contidos em superestados desenhados verticalmente são desenhados horizontalmente, e vice-versa. Diagramas desenhados utilizando a função de layout automático das ferramentas Rational Rose e ViSta foram comparados pelos autores. Os desenhos gerados pela ferramenta ViSta apresentaram menos cruzamentos entre arestas, menor número de curvas nas arestas, menor área e uma boa relação de aspecto.

Existe um editor gráfico de diagramas em Xchart, chamado Smart. Esse editor foi desenvolvido como parte de uma dissertação de mestrado por Osvaldo Severino Jr. [34]. Implementado para a plataforma Windows, o editor permite a captura de diagramas exclusivamente na forma da árvore de estados correspondente ao diálogo. As transições são então adicionadas à árvore, criando uma representação equivalente ao modelo Xchart. Essa representação pode ser transformada pelo editor, exclusivamente para visualização, em um diagrama aninhado representado usando-se a sintaxe Xchart.

2.4.2 Bibliotecas e frameworks

Na construção de editores de diagramas, vários padrões de desenvolvimento e implementação de recursos se repetem, sendo comuns à maioria desses sistemas. Para facilitar a construção de tais editores, foram criadas bibliotecas e frameworks com esse fim. Esses componentes são ligados aos editores e oferecem recursos como a visualização de figuras, a gerência da interação com o usuário e implementações de algoritmos de layout automático.

Eichelberger et al. [35] estenderam o framework SugiBib para o layout de diagramas UML. O framework SugiBib é um conjunto de componentes escritos em Java que, combinados em seqüências específicas, implementam o algoritmo de Sugiyama ou outros algoritmos de layout de grafos. Os diagramas UML foram representados como grafos em que vértices são elementos como pacotes e classes e arestas são generalizações, associações, agregações, composições e dependências.

GEF é um framework para o desenvolvimento de editores de diagramas na forma de plugins para o Eclipse IDE [36]. Dividido em dois componentes, Draw2d (para a visualização de diagramas e gráficos vetoriais em geral) e GEF propriamente dito (responsável pelo controle da edição dos diagramas), é implementado em Java e integra-se fortemente à plataforma Eclipse. É organizado segundo o padrão de projeto MVC (Modelo-Visão-

Controlador) e oferece funções como paleta de ferramentas, gerenciamento de elementos e conexões do diagrama, exibição e modificação de propriedades, zoom, gerência de comandos com funcionalidade de *undo* e *redo* e algoritmos simples de layout automático.

2.4.3 Geração de código

Uma alternativa à utilização de bibliotecas e frameworks é a geração automatizada de código para a criação de editores de diagramas. Nessas ferramentas, a entrada é a especificação da linguagem visual dos diagramas a serem editados, incluindo suas características gráficas (nível léxico) e as combinações possíveis entre os elementos (nível sintático). Como saída, a ferramenta gera o código fonte de um editor capaz de editar tais diagramas, com maior ou menor número de funcionalidades como verificação de consistência sintática, layout automático e tradução de diagramas de e para linguagens alternativas ou textuais.

DiaGen [37, 38] é uma ferramenta para a geração de editores de diagramas a partir de especificações de linguagens visuais. A ferramenta foi escrita em Java e os editores gerados também são criados nessa linguagem. A especificação da linguagem visual, que envolve sua sintaxe e sua representação gráfica, é bastante simplificada porque se restringe ao domínio dos diagramas similares a grafos. Esses diagramas são sempre compostos de vértices e arestas que conectam vértices. Os vértices também podem ser hierárquicos, isto é, conter outros grafos; nesse caso, as arestas podem cruzar os limites dos vértices. Tanto os vértices quanto as arestas podem ser divididos em categorias (exemplo: arestas de um diagrama UML podem ser generalizações, associações, etc.) cujo relacionamento é descrito de forma declarativa, segundo a especificação passada à ferramenta.

O DiaGen consiste em um framework de edição de diagramas e um gerador de programas. O gerador de programas, a partir de uma especificação de linguagem visual, cria um conjunto de classes Java. Essas classes, unidas ao framework de edição de diagramas, formam o editor de diagramas para a linguagem especificada. Os editores gerados permitem o desenho livre de diagramas, dando total controle ao usuário sobre a edição do diagrama, e um modo de edição assistida, em que cada modificação é orientada pela sintaxe da linguagem e o layout do diagrama é ajustado automaticamente.

No modo de desenho livre o editor analisa a sintaxe do diagrama depois de cada modificação, destacando as partes que estão sintaticamente corretas. Essa verificação é feita em 4 etapas sucessivas: scanning, redução, parsing e avaliação de atributos. Na etapa de scanning um modelo hipergrafo do diagrama (conhecido como HGM, hypergraph model) é criado, ao mapear cada vértice e aresta do diagrama em hiperarestas que são conectadas aos pontos de contato de vértices e arestas de acordo com as relações espaciais entre os elementos diagrama (contato entre vértice e aresta e composição de vértices hierárquicos). No passo de redução o HGM de um diagrama, que costuma ser bastante grande mesmo

para diagramas pequenos, um modelo hipergrafo reduzido (rHGM) é criado a partir da aplicação de transformações ao HGM original que identificam as informações necessárias sobre o diagrama, num processo semelhante à análise léxica de um compilador. Na etapa de parsing um parser de hipergrafos, incluído em todo editor gerado pelo DiaGen e baseado numa gramática livre de contexto obtida a partir da especificação da linguagem de diagramas fornecida, analisa a sintaxe do rHGM, encontrando os possíveis erros de sintaxe no diagrama desenhado pelo usuário. Na fase de avaliação de atributos a representação sintática do diagrama é aumentada com atributos, dando origem a uma representação semântica. Essa representação pode então ser usada, por exemplo, para a aplicação de um algoritmo de layout automático. A ferramenta DiaGen não é mantida desde 2004 e o site que a hospedava não está mais ativo.

Outra ferramenta para geração de editores de diagramas é a Penguins, proposta por Chok [28], que utiliza gramáticas de restrições à coleções (*CMG – Constraint Multiset Grammars*) para definir a linguagem visual dos diagramas. Esse formalismo lógico possibilita a descrição da sintaxe de linguagens visuais, tanto para geração quanto para reconhecimento de diagramas. Uma CMG é composta por regras na forma $P ::= P_1, ..., P_n where C$, onde P é um símbolo não terminal reconhecido da coleção $P_1, ..., P_n$ dada a restrição C. Os símbolos terminais são primitivas gráficas como linhas e círculos e restrições descrevem a relação geométrica existente entre essas primitivas. A partir dessas gramáticas, o editor gerado utiliza *parsing* incremental e um resolvedor de restrições para verificar a consistência do diagrama desenhado e corrigir erros do usuário durante a edição. Os editores são gerados na linguagem C++, mostrando bom desempenho na análise sintática dos diagramas durante a edição. Trabalhos futuros planejados incluem a possibilidade de especificar parâmetros para o layout automático dos diagramas e regras para simulação e transformação de modelos.

GMF (*Graphical Modeling Framework*) é um projeto da fundação Eclipse para a criação de um framework destinado à geração de editores de diagramas, combinando modelos gerados pela ferramenta EMF (*Eclipse Modeling Framework*) e as funcionalidades oferecidas pelo GEF [39]. No GMF, um metamodelo EMF é combinado com definições sobre a aparência gráfica dos seus elementos e as operações possíveis sobre eles para a geração de um plugin de um editor de diagramas para a plataforma Eclipse capaz de editar os modelos especificados. Ainda em desenvolvimento, essa solução se mostra imatura para a utilização em larga escala e tem como principal desvantagem a limitação da linguagem visual pelo metamodelo EMF utilizado, com o qual a sintaxe da linguagem é fortemente ligada. Essa limitação dificulta a verificação da sintaxe do diagrama editado, exigindo outras formas menos apropriadas de especificação e checagem de sintaxe, como a linguagem OCL (*Object Constraint Language*), com a qual é possível definir restrições sobre o modelo e, indiretamente, descrever as construções válidas do diagrama.

Essa limitação motivou a criação de um projeto independente, Tiger [39, 40], que assim como o GMF combina modelos implementados em e gerenciados pelo EMF e a geração de editores de diagramas para a plataforma Eclipse. Para superar as limitações do GMF, o projeto Tiger substitui a ligação da sintaxe dos diagramas ao metamodelo EMF pela transformação de grafos a partir de uma gramática formada por regras de reescrita de grafos. O alfabeto utilizado pela gramática inclui vértices e arestas tipados que podem conter atributos e também define sua aparência gráfica. Dessa forma, tanto regras de edição quanto, futuramente, regras para simulação e transformação de modelos podem ser especificadas. Os editores gerados têm edição orientada à sintaxe e são capazes apenas de editar diagramas simples baseados em grafos dirigidos.

2.5 Layout automático de diagramas

O layout automático é um recurso importante nos editores de diagramas. Consiste em reposicionar e redimensionar os elementos de um diagrama existente, com o objetivo de melhorar sua legibilidade e, assim, liberar o usuário da tarefa de ajustar o layout manualmente (sem o auxílio de sistemas automatizados que posicionem os elementos do diagrama). Isso é feito buscando atender a uma série de requisitos, listados na seção 2.5.1, através de algoritmos como os listados na seção 2.5.2.

2.5.1 Características desejáveis

Várias características supostamente importantes para a legibilidade de grafos foram consideradas na proposta de algoritmos de layout automático. Vários dos algoritmos propostos buscam minimizar:

- os cruzamentos de arestas (definidos como a intersecção entre as linhas ou curvas que representam arestas de um grafo);
- o comprimento das arestas;
- as dobras em arestas (mudanças de direção na representação das arestas, usadas para evitar a sobreposição com outros elementos do grafo);
- a área total do layout;
- a largura do layout.

Alguns dos algoritmos procuram maximizar:

- o ângulo mínimo entre arestas de um mesmo vértice (distribuindo-as o mais igualmente possível ao redor do elemento);
- a ortogonalidade (alinhamento entre elementos do grafo numa mesma grade ortogonal);
- a simetria (distribuição uniforme dos elementos do grafo).

Finalmente, existem algoritmos que tentam aproximar de 1 o *aspect ratio* (relação entre a largura e a altura da representação visual) do grafo.

Purchase fez uma série de estudos sobre o desempenho de algoritmos de layout automático quanto à sua legibilidade, tanto em relação à grafos em geral quanto com relação à diagramas de domínios específicos, como diagramas UML [41, 42, 43]. Segundo esses estudos, existe um número de características ou estéticas que tornam diagramas mais legíveis e que deveriam ser buscadas nos algoritmos de layout automático. A principal dessas estéticas é a minimização dos cruzamentos entre arestas, seguida pela minimização de dobras em arestas e maximização da simetria. A ortogonalidade e a maximização do ângulo mínimo não apresentaram melhoras significativas na legibilidade, mas a primeira é apreciada pelos usuários. Além disso, rótulos de vértices e arestas devem ter a mesma orientação (preferencialmente todos horizontais ou todos verticais) e usar uma única fonte de texto. A largura de diagramas que contêm grande quantidade de texto nos rótulos deve ser adequada para acomodar a informação representada. Por fim, o fluxo natural dos grafos direcionais, representado pelo sentido das arestas, deve ser valorizado.

Marey [44] é parte da ferramenta de desenho de grafos JJGraph, descrita e implementada por Friedrich e Eades. Marey implementa algoritmos para a animação de grafos com o propósito de facilitar o acompanhamento, pelo usuário, de modificações estruturais nos grafos, aumentando a usabilidade do editor JJGraph.layout incremental Objetivo dos algoritmos é preservar o mapa mental e comunicar as mudanças estruturais no grafo durante a animação, atendendo a critérios estéticos como facilidade de seguir o movimento de vértices e arestas, estruturação do movimento do grafo, transição suave da origem para o destino, evitar desenhar estruturas inexistentes, etc. A animação é realizada em quatro etapas: esconder vértices e arestas removidos, movimentação rígida, interpolação linear e mostrar novos elementos. Na fase de movimentação rígida, o grafo é movimentado como se fosse uma estrutura tridimensional rígida até o mais próximo possível da posição final. Na etapa de interpolação linear, os vértices que ainda não estão na posição final são movidos linearmente para as posições corretas. Um protótipo da implementação foi integrado no framework InVision.

DynaDAG [45], descrita por North, é uma nova heurística para o layout incremental de grafos dirigidos acíclicos. As operações sobre grafos previstas pelo DynaDAG são a

inserção, a remoção e a otimização de vértices e arestas. Estabilidade geométrica (posição dos elementos) é mantida precisamente, enquanto que a estabilidade topológica (ordem dos elementos) é mantida heuristicamente. Segundo essas heurísticas, vértices são mais estáveis do que arestas. Quando um vértice é inserido uma nova camada é escolhida, vértices e arestas pré-existentes são ajustados de acordo com essa escolha e finalmente o vértice é inserido, com otimizações locais de sua posição e de suas arestas. DynaDAG, assim como outros algoritmos de layout, foi implementado na biblioteca DynaGraph. Visualizadores gráficos que utilizam essa biblioteca foram criados para as plataforma Windows e Unix Tcl/Tk.

2.5.2 Algoritmos

Existem algoritmos conhecidos para entradas que são limitadas quanto à sua conexidade (como aqueles específicos para árvores), quanto ao grau máximo dos vértices, quanto à existência de direção nas arestas ou ciclos, quanto à planaridade e quanto à estrutura (como os voltados à grafos compostos ou hierárquicos).

Entre as métricas priorizadas podemos destacar como foco de vários algoritmos a minimização de cruzamentos entre arestas ou das dobras em uma mesma aresta (objetivando a melhoria da legibilidade), da área ocupada pelo mapa (principalmente em aplicações de projeto de circuitos digitais), assim como o fluxo de informações ou hierarquia representados e a estrutura global do grafo (características valorizadas em diagramas da área de Tecnologia da Informação [46]).

Quanto às características do mapa gerado, alguns tipos de mapas são recorrentes (como os mostrados na figura 2.2). Mapas ortogonais organizam seus vértices em uma matriz e suas arestas são representadas como seqüências de segmentos de reta horizontais e verticais, minimizando a área ocupada pelos vértices. Mapas orgânicos não se preocupam com o alinhamento dos vértices, priorizando a simetria do mapa e a estrutura do grafo ao posicionar os vértices livremente e desenhar arestas como segmentos sem curvas ou dobras. Mapas circulares posicionam os vértices ao longo de circunferências, dando grande visibilidade aos ciclos do grafo. Mapas em camadas dispõem os vértices em camadas sucessivas, alinhando os vértices de uma mesma camada; a hierarquia ou o fluxo de informações representados pelo grafo são valorizados.

Quanto à técnica usada no algoritmo, são comuns a simulação iterativa de forças físicas como repulsão e atração, preferido na geração de mapas orgânicos, e o posicionamento baseado em heurísticas, mais utilizados em mapas ortogonais ou em camadas.

A escolha do algoritmo a ser aplicado no layout de um grafo depende do domínio da aplicação, isto é, deve levar em conta as propriedades dos grafos ou diagramas e as características valorizadas pelos usuários que os criarão ou interpretarão.

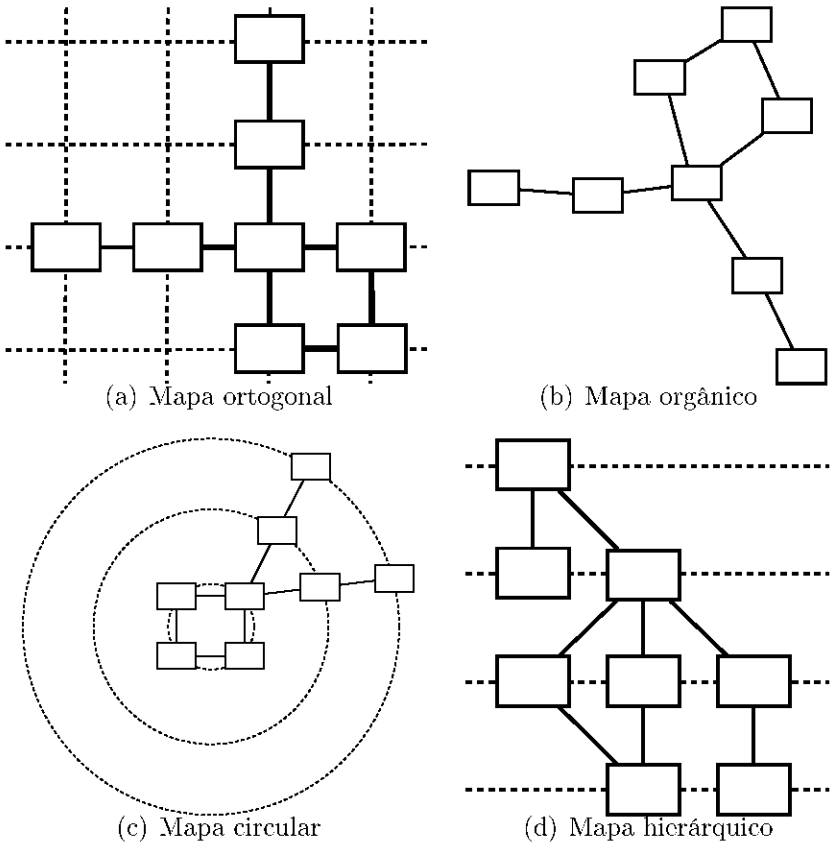


Figura 2.2: Tipos de mapas quanto às suas características

Misue *et al.* propõe dois algoritmos de ajuste de layout que buscam manter o mapa mental do usuário sobre um grafo ou diagrama [47]. O mapa mental é definido sobre três conceitos, que devem ser mantidos entre dois layouts para que se diga que o mapa mental foi mantido: ordenação ortogonal (preserva a posição relativa entre dois vértices), proximidade (distância geométrica entre vértices do diagrama) e topologia (posição de vértices em relação ao layout como um todo). O primeiro algoritmo proposto busca ajustar o layout em que vértices se sobrepuseram indevidamente através da simulação de forças de repulsão entre os vértices sobrepostos. Outra técnicas descritas incluem a ordenação de vértices e o escalonamento uniforme. Também são analisados algoritmos de layout apropriados para grafos muito grandes, em que uma parte deste é apresentada em detalhes e o resto do grafo é representada em proporção menor, segundo o critério de manutenção de mapa mental.

2.5.2.1 Simulação de forças físicas

Eades e Huang [48] descrevem a arquitetura CGA, um sistema para visualização interativa de grafos aninhados. CGA é uma arquitetura em camadas (grafos, clustering, resumo e figura), para que o sistema hospedeiro não precise conhecer o grafo inteiro (que pode ter milhões de vértices e arestas), agentes externos como algoritmos de layout e de desenho de grafos possam ser utilizados e cada camada possa ser especializada em seus serviços. A camada de grafos oferece primitivas para manipulação de grafos (inserção e remoção de vértices e arestas), a de clustering para manipulação de clusters (criação e destruição de clusters), a de resumo para manipulação do resumo do grafo (expansão e colapso de clusters) e a de figura para visualização e animação de grafos (movimentação de vértices, escalonamento e operações de layout). O layout é aplicado com a simulação de forças físicas, divididas em três categorias: internas (entre vértices de um mesmo cluster), externas (entre vértices de clusters diferentes) e virtuais (entre o centro de cada cluster e os vértices que este contém). Cada operação sobre o grafo é acompanhada de animações que combinadas representam a operação e as mudanças no layout. A arquitetura CGA foi implementada no sistema DA-TU, que oferece todas as operações descritas.

O algoritmo de Fruchterman e Rheingold foi adaptado por Enright et al. [49] para o layout de grafos de similaridades biológicas usando como estudo de caso as similaridades existentes entre proteínas, cujos grafos costumam ter muitas arestas para cada vértice. Resumidamente, o algoritmo consiste em dispor os vértices aleatoriamente para que estes sejam deslocados pela simulação iterativa de atrações e repulsões entre eles. Essas atrações e repulsões são proporcionais a valores associados às arestas do grafo (similaridade entre proteínas, por exemplo). O deslocamento possível aos vértices é dado por uma função temperatura que, inicialmente, tem valor alto. Ao longo da simulação, o valor da função diminui até atingir um mínimo constante. O algoritmo resultante, apesar de não ser

ótimo, é suficiente para o layout esteticamente agradável dos grafos utilizados, além de ter bom desempenho. A implementação do algoritmo permite a geração de visualizações em duas ou três dimensões, sendo a última representação explorável interativamente.

2.5.2.2 Disposição de elementos

A ferramenta daVinci [50], criada na Universidade de Bremen, permite a visualização interativa de grafos controlada por uma aplicação cliente. Através de comunicação bidirecional com a aplicação, a ferramenta exibe grafos cíclicos ou acíclicos a partir de representações textuais que, além dos vértices e arestas do grafo, contém atributos dos vértices (como textos, fontes, cores e formas). O algoritmo de layout da ferramenta é baseado no algoritmo de Sugiyama, com modificações, que entre outras coisas, diminuem o número de cruzamentos de arestas. Também é usado um algoritmo de layout de árvores de complexidade linear baseado no trabalho de Juustistenaho [51], que diminui o número de passagens em uma árvore de duas para uma. Entre as funções interativas da ferramenta, disponíveis após a aplicação inicial do algoritmo de layout, estão o ajuste fino (reposicionamento de vértices e arestas, mantido em novas aplicações do algoritmo de layout), abstrações interativas (possibilidade de esconder o subgrafo ou as arestas de um vértice) e escalonamento. O estado da visualização pode ser salvo em arquivo para futuras interações ou exportado para o formato PostScript para impressão. São usados dois pipes unidirecionais para a comunicação da ferramenta com a aplicação cliente. O primeiro, no sentido aplicação-cliente, é usado para transferir grafos, adicionar menus à interface de usuário da ferramenta, iniciar diálogos com o usuário, mostrar informações sobre estado ou controlar o sistema de visualização. O segundo, de sentido oposto, é usado para informar a aplicação sobre eventos gerados pelo usuário, como seleção de vértices ou arestas, ativação de menus e respostas à diálogos iniciados pela aplicação.

Gansner et al. [52] descreveram o sistema dag, uma implementação de algoritmos de layout automático de grafos dirigidos, direcionada a grafos acíclicos. Essa ferramenta cria, a partir de uma representação textual de um grafo, uma representação também textual do layout do grafo. Esse modo de operação facilita a integração do dag com outros sistemas. Exemplos dessa integração são a visualização de desempenho de programas (analisados pela ferramenta gprof), estrutura de programas C/C++ (obtida das ferramentas cia, cia++, dagen, incl, ciatso, etc.), máquinas de estado finitas, chamadas de função e estruturas de dados (interativamente, com a ferramenta Xray). dag é otimizada para grafos dirigidos acíclicos, onde se pode notar um fluxo de um vértice a outro do grafo, e usa quatro passos para a geração do layout dos grafos: atribuição de camadas, ordenação de vértices, coordenadas de vértices e desenho de arestas. Na fase de atribuição de camadas os vértices do grafo, transformado são divididos em camadas seqüenciais e adjacentes, através de programação inteira, com o objetivo de diminuir o comprimento

das arestas desenhadas e manter o fluxo do grafo. Na etapa de ordenação de vértices, os vértices de cada camada são ordenados, através de heurísticas, para minimizar o número de cruzamentos de arestas. As coordenadas dos vértices são calculadas, também por programação inteira, para minimizar a distância horizontal entre os vértices. Na última etapa, de desenho de arestas, estas são representadas por splines cujas curvas desviam dos vértices e arestas já desenhados.

Tatzmann [53] descreve um algoritmo, também baseado no trabalho de Sugiyama, que permite a visualização de grafos com ênfase em um vértice central qualquer, circundado pelos outros vértices do grafo. Também há a possibilidade de navegação entre os vértices do grafo, com a escolha de um novo vértice central. A mudança de vértice central é animada, usando-se para isso um algoritmo de desenho dinâmico de grafos que movimenta agrupamentos de vértices como objetos rígidos. Essas técnicas foram implementadas como uma extensão ao pacote de desenho de grafos JMFGraph (Java Modular Framework for Graph Drawing).

Eades e Feng [54] descrevem algoritmos para o desenho de grafos aninhados e propõem algoritmos para o desenho tridimensional desses grafos. Dado um grafo $C = (G, T)$ onde G é um grafo não dirigido e T é uma árvore cujas folhas são os vértices de G, uma representação planar (possivelmente abstrata) de G equivalente a cada nível de T é criada. As representações dos clusters de cada nível são ligadas por arestas com as representações dos clusters ou vértices respectivos nos níveis superior e inferior. Essa representação tridimensional fornece uma melhor visão do modelo mental associado a grafos aninhados complexos.

2.5.2.3 Algoritmo de Sugiyama

O algoritmo de Sugiyama originalmente [55] tem como entrada dígrafos acíclicos, gera mapas em camadas e prioriza a minimização de cruzamentos entre arestas e do seu número de dobras, procurando destacar a hierarquia representada pelo grafo. Devido ao sucesso alcançado na valorização dessas métricas, esse algoritmo é largamente aplicado no layout de grafos dirigidos em geral. Foi melhorado e adaptado, inclusive para o layout de dígrafos compostos [33].

Esse algoritmo é composto de três etapas básicas, como mostrado na figura 2.3: atribuição de camadas, redução de cruzamentos entre arestas e atribuição de coordenadas.

A atribuição de camadas é subdividida em três partes, iniciando com uma fase de pré-processamento do grafo para desfazer os ciclos existentes com a inversão do sentido de arestas. Após esse pré-processamento o grafo é ordenado topologicamente, atribuindo níveis aos vértices a partir das fontes do grafo, particionando-o. Na terceira fase, vértices fictícios são inseridos em cada cruzamento entre uma aresta e um nível, dividindo arestas longas em um caminho de arestas menores. Dessa forma, toda aresta conecta vértices de

1. Atribuição de camadas
2. Eliminação de ciclos
   (a) Ordenação topológica
   (b) Inserção de vértices fictícios
3. Redução de cruzamentos entre arestas
4. Atribuição de coordenadas

Figura 2.3: Algoritmo de Sugiyama

camadas imediatamente sucessivas, tornando o grafo uma hierarquia própria.

Na etapa de redução de cruzamentos os vértices de cada camada são reordenados de acordo com o baricentro dos seus vizinhos da camada seguinte. Dada uma camada m_i e sua camada sucessiva m_{i+1}, a cada vértice $v \in m_{i+1}$ é atribuído um valor correspondente ao baricentro de seus vizinhos (onde $p(x)$ é a posição de x na sua camada e $n(v)$ é o conjunto dos vizinhos incidentes em v):

$$b(v) = \begin{cases} \sum_{x \in n(v)} \frac{p(x)}{|n(v)|} p(x) & \leftarrow \quad |n(v)| > 0 \\ p(x) & \leftarrow \quad |n(v)| = 0 \end{cases}$$

Com essa informação os vértices da camada m_{i+1} são ordenados em ordem crescente pelo valor dos baricentros de seus vizinhos da camada m_i. Esse processo inicia-se com $i = 1$ e segue até o último par de camadas para repetir-se no sentido inverso, ordenando vértices da camada m_i a partir dos baricentros dos vértices da camada m_{i+1}. Essa etapa é repetida até que nenhuma mudança na ordem dos vértices dentro de cada camada seja detectada ou até que um limiar pré-definido de iterações seja atingido.

Na etapa de atribuição de coordenadas, cada vértice recebe uma coordenada horizontal, sempre respeitando a ordem dos vértices dentro de cada camada estabelecida na etapa anterior. A cada vértice é atribuída uma prioridade: máxima para os vértices fictícios ou correspondente ao grau para os outros vértices. Cada vértice é posicionado o mais próximo possível do baricentro de seus vizinhos, movimentando vértices em ordem crescente de prioridade.

2.5.2.4 Algoritmo do editor Smart

Uma das extensões do algoritmo de Sugiyama permite a aceitação de dígrafos compostos como entrada. No editor Smart [34] esse algoritmo foi adaptado para o layout de

diagramas Xchart. Entre as adaptações feitas estão a remoção de arestas que quebram a condição de adjacência, isto é, arestas conectando um vértice e seus descendentes ou ascendentes, e modificações na etapa de atribuição de coordenadas para considerar construções específicas da linguagem Xchart. Como as diferenças entre ambos os algoritmos são pequenas, será descrito o algoritmo implementado no Smart.

Uma árvore de inclusão é uma árvore $D_I = (V, E)$ com raiz fixa $r \in V$. Dado um vértice $v \in V$ qualquer e o caminho $rx_2x_3...w...x_{n-1}v$ de r a v, diz-se que $w \neq r$ é descendente de r e $w \neq v$ é ancestral de v. A raiz da árvore não tem ancestrais e as folhas não têm descendentes.

Um dígrafo de adjacência $D_A = (V, F)$ é um dígrafo simples no qual todas as arestas conectam vértices adjacentes. Um vértice u é adjacente a v se não é ancestral nem descendente de v, ou seja, $v \notin An(u) \cup De(u)$. Assim, $uv \in F \rightarrow adj(u, v)$.

Diagramas Xchart podem ser vistos como dígrafos compostos. Um dígrafo composto é formado por uma tupla $G = (V, E, F)$ onde V é o conjunto de vértices, E é o conjunto de arestas da árvore de inclusão e F é o conjunto de arestas do dígrafo de adjacência. Dessa forma, um dígrafo composto une em um mesmo grafo a composição de vértices em subgrafos e as arestas que conectam esses vértices.

O algoritmo de layout do editor Smart tem como entrada um dígrafo composto equivalente ao diagrama Xchart a ser visualizado. Esse algoritmo é dividido aproximadamente nos mesmos passos do algoritmo original de Sugiyama, como mostrado na figura 2.4.

1. Hierarquização
2. Substituição de arestas de adjacência
   (a) Composição de níveis para os vértices
   (b) Reversão da orientação de arestas adjacentes
3. Normalização
4. Ordenação de vértices
5. Layout métrico

Figura 2.4: Algoritmo de layout do editor Smart

A etapa de hierarquização atribui um nível para cada vértice do grafo; vértices de mesmo nível serão desenhados um ao lado do outro no mapa, alinhados verticalmente. Essa etapa é subdividida em três fases. Inicialmente, as arestas de adjacência que conectam vértices de profundidade diferente na árvore de inclusão são mapeadas para arestas

especiais que conectam vértices de mesma profundidade. A seguir, a cada vértice v cuja profundidade é $d(v)$, é atribuído um valor $l(v) = c_1c_2...c_{d(v)}$ composto pelos valores dos seus ancestrais e um inteiro, de forma que vértices cujos filhos se conectam tenham o mesmo valor. Após isso, arestas $uv \in F$ tal que $l(u) \geq l(v)$ são invertidas.

Na etapa de normalização, vértices fictícios são inseridos entre toda aresta uv em que $d(u) \neq d(v)$ ou $x_{d(u)} \neq y_{d(v)}$, com $l(u) = x_1x_2...x_{d(u)}$ e $l(v) = y_1y_2...y_{d(v)}$. Com a normalização do dígrafo todas as arestas de adjacência conectam vértices de mesma profundidade e são evitados cruzamentos entre arestas e os limites de retângulos.

A etapa de ordenação de vértices é similar à etapa de redução de cruzamentos entre arestas do algoritmo original de Sugiyama. Os vértices de cada camada são ordenados iterativamente de acordo com o baricentro de seus vizinhos até que não existam mudanças ou um limiar máximo seja alcançado.

A etapa de layout métrico é equivalente à etapa de atribuição de coordenadas do algoritmo original, com o posicionamento dos vértices obtido através do método de baricentros com prioridades.

2.6 Outros conceitos

2.6.1 Manipulação direta

Manipulação direta é um estilo de interface onde os objetos de interesse, incluindo os seus estados, são representados continuamente e modificados através de ações rápidas, reversíveis e incrementais [56]. No caso de um editor de diagramas, este oferece visões de partes do diagrama, sempre de forma a ressaltar aspectos relevantes, do ponto de vista do usuário do editor, do objeto editado. A cada momento existe um conjunto de ferramentas disponíveis para a manipulação do diagrama ou de uma de suas visões. Através da seleção de elementos do diagrama e da ativação de ferramentas, o diagrama e suas visões são manipulados até que o diagrama desejado pelo usuário seja obtido.

Para a iniciar uma operação o usuário deve comunicar à aplicação, por meio da interface, pelo menos dois atributos da operação: o objeto a ser transformado, a transformação a ser efetuada e, opcionalmente, parâmetros para a operação.

A seleção dos objetos a serem transformados se dá com a especificação dos atributos que os objetos selecionados devem ter, obtendo-se um subconjunto do conjunto de objetos selecionáveis. Nas interfaces de manipulação direta, em geral, e nos editores de diagramas, em particular, essa seleção costuma se basear na posição dos objetos gráficos de interesse do usuário. A definição dos pontos ou da região que contêm os objetos de interesse se dá de forma mais natural com o auxílio de um dispositivo apontador, como *mice* e mesas digitalizadoras. Esses dispositivos são usados para indicar um ponto dentro da região

definida por um objeto ou para definir os pontos que delimitam a região que contém os objetos de interesse.

Para a ativação das operações existem três ações padronizadas no ambiente em que o editor proposto será executado: clique em botões de barras de ferramentas, ativação de ítens de menu e pressionamento de combinações de teclas [57]. Barras de ferramentas são grupos de botões, usualmente icônicos e alinhados horizontalmente ou verticalmente, que representam um subconjunto de operações mais freqüentes dentro do conjunto de operações disponíveis numa aplicação. Menus são grupos de botões, também alinhados verticalmente ou horizontalmente, que representam textualmente as operações disponíveis numa aplicação. Menus podem ser aninhados hierarquicamente, quando a ativação de um item de menu resulta na exibição de um submenu previamente invisível, e assim sucessivamente. Essa propriedade dos menus, aliada ao seu caráter textual/simbólico, possibilita o acesso à mais operações do que o permitem as barras de ferramentas. O pressionamento de combinações de teclas para a ativação de operações é considerado um atalho direcionado aos usuários avançados de uma aplicação. Apesar das combinações de teclas disponíveis serem muitas vezes exibidas junto aos ítens de menu e botões de barras de ferramentas correspondentes, sua natureza pouco visual e a exigência de memorização sugerem uma utilização moderada dessa forma de ativação de operações.

Os parâmetros das operações realizadas podem, também, ser especificados através de dispositivos apontadores ou por meio de processos mais complexos de interação, como o de entrada e modificação de texto. No primeiro caso estão, por exemplo, a indicação da posição em que um objeto deve ser criado ou para onde este deve ser movimentado. A modificação dos rótulos associados a elementos do diagrama é um exemplo do segundo caso, onde a modificação de texto oferece a flexibilidade necessária ao projetista usuário do editor.

2.6.2 WIMP

WIMP é um acrônimo para "windows, icons, menus, pointing device" (janelas, ícones, menus e dispositivo apontador) [58]. É um estilo de interação para interfaces de usuário desenvolvido nos anos 70 no centro de pesquisas de Palo Alto, da Xerox. Os elementos que compõem o seu nome são as principais inovações desse estilo de interação, tornadas possíveis devido à disponibilização de telas gráficas e a invenção do mouse.

Janelas são regiões da tela reservadas para a exibição dos dados de interesse do usuário. Ícones representam, através de pequenas imagens desenhadas na tela, dados ou componentes do sistema. Menus são listas de operações disponíveis ao usuário, descritas textualmente ou graficamente, e agrupadas em regiões da tela. Um dispositivo apontador, geralmente um mouse, permite a indicação, pelo usuário, de pontos na tela simultanea-

mente ao pressionamento de botões que podem ser detectados pelo sistema. O dispositivo apontador é usado para interagir com os outros elementos do estilo WIMP: se o ponto selecionado pelo dispositivo, indicado por um desenho característico (cursor), estiver contido em uma janela, ícone ou item de menu, este elemento é ativado e a operação que descreve é iniciada.

Entre as vantagens do estilo WIMP estão a facilidade de aprendizado na utilização de uma interface de usuário, a acessibilidade tanto a usuários iniciantes quanto avançados (por exemplo, através de atalhos) e a grande padronização entre as interfaces, passíveis de uso por lembrança (sem a necessidade de um novo aprendizado). Por outro lado, interfaces WIMP apresentam dificuldade para a especificação de operações complexas ou repetitivas, exigem um esforço significativo de interação com a interface (em oposição à interação com o sistema) e dificultam a manipulação de elementos tridimensionais.

2.6.3 Padrões de projeto

Um padrão de projeto é uma codificação de uma solução para um problema recorrente no projeto de softwares. Cada padrão inclui a definição do problema, as circunstâncias que o criam e a solução proposta [59]. Uma série de padrões clássicos foi descrita por Gamma et al. [60] em 1995. Entre os padrões usados na implementação do editor Sins estão o padrão Modelo-Visão-Controlado, Fábrica, Observador e Comando, descritos a seguir.

2.6.3.1 Modelo-Visão-Controlador

O padrão Modelo-Visão-Controlador é uma solução para o problema de apresentar informações complexas ao usuário. Mais especificamente, o padrão MVC desacopla o modelo dos dados a serem apresentados da visão (correspondente à interface de usuário usada para a apresentação dos dados). Dessa forma é possível modificar a estrutura dos dados sem que a interface seja alterada e vice-versa.

Esse desacoplamento entre o modelo e a visão é realizado com a utilização de um objeto intermediário, chamado controle. O controle é responsável por gerenciar a visão que representa uma parte do modelo e atualizar o modelo de acordo com os eventos gerados pelo usuário na interação com a visão.

O comportamento de um sistema no padrão MVC pode ser descrito por um ciclo:

1. O usuário interage com a interface de usuário correspondente à visão, gerando um evento que é capturado pelo controlador.

2. O controlador responde ao evento modificando o modelo.

3. O modelo modificado repassa as alterações à visão, para que esta possa atualizar sua representação do modelo.

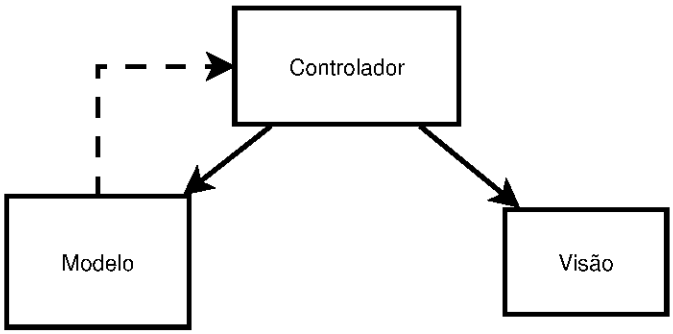


Figura 2.5: Padrão de projeto MVC

Uma variação do padrão MVC é o MVC Hierárquico (*HMVC — Hierarchical MVC*) [61]. No HMVC uma hierarquia de agentes, cada um implementando o padrão MVC, é utilizada para implementar interfaces de usuário complexas. Nesse caso, a comunicação entre os agentes se dá através dos seus controladores.

2.6.3.2 Fábrica

O padrão Fábrica inclui duas variações, ambas para o problema de criar objetos sem especificar suas classes: Método Fábrica (*Factory Method*) e Fábrica Abstrata (*Abstract Factory*).

Um método fábrica declarado em uma classe ou interface tem a responsabilidade de criar objetos, normalmente retornando ao método que o chamou uma referência (concreta ou abstrata) para o objeto criado. Assim, classes que implementam esse método podem criar objetos de diferentes tipos, sempre através da mesma interface.

Uma fábrica abstrata é uma classe que contém métodos fábrica para a criação de objetos de tipos relacionados, retornando referências abstratas para os objetos criados. Os objetos clientes utilizam os serviços do objeto fábrica (uma instância concreta da fábrica) através de sua interface abstrata. Dessa forma, os clientes podem usar implementações concretas diferentes da fábrica, criando as instâncias concretas de objetos que são mais adequadas ao seu propósito mesmo sem conhecer os detalhes da sua criação.

2.7 Conclusão

Sistemas reativos, presentes em muitas áreas da indústria e da computação, são aqueles em que existe uma interação contínua entre o sistema e o seu ambiente. Foi definido que um sistema interativo é um sistema reativo composto por uma camada computacional e uma camada interativa, com a qual o usuário se comunica. Essa camada interativa, ou

interface de usuário, pode ser estruturada em três componentes: apresentação, controle de diálogo (também um sistema reativo) e interface de software.

A linguagem Xchart, descrita em detalhes no capítulo 3, assim como sua ancestral Statechart, é classificada como um modelo de redes de transição e pode ser usada para a especificação de sistemas reativos dentro de uma arquitetura orientada a modelos. Nessa linguagem, o comportamento de um sistema é descrito por um diagrama de máquina de estados estendido com construções que aumentam seu poder de expressão, permitindo por exemplo a modularização e a descrição de estados concorrentes.

As vantagens da linguagem Xchart são potencializadas com a utilização de um ambiente de desenvolvimento que apóie a descrição e depuração de sistemas reativos. Um dos principais componentes de tal ambiente é um editor Xchart com recursos para a edição de especificações Xchart. O editor existente, Smart, não é capaz de atender à todas as necessidades correntes dos desenvolvedores de sistemas reativos. Por isso, no capítulo 4, os requisitos de um novo editor são explorados.

Capítulo 3

Tecnologia Xchart

3.1 Introdução

A tecnologia Xchart tem por objetivo permitir a modelagem e implementação de componentes de controle de diálogo de interfaces de usuário, podendo também ser usada para a construção de sistemas reativos em geral. Essa tecnologia engloba vários elementos envolvidos nesse processo de construção. Entre eles estão um ambiente de apoio à modelagem desses sistemas, descrito na seção 3.2, a linguagem visual Xchart, usada para descrever esses sistemas, descrita na seção 3.3, e a linguagem TEXchart, a equivalente textual da linguagem Xchart, descrita na seção 3.4.

3.2 Ambiente de apoio

O ambiente de apoio da plataforma Xchart divide-se em dois: um ambiente de design de interfaces de usuário, para apoiar a descrição e implementação de sistemas de controle de diálogo, e um sistema de execução, responsável pela execução dos sistemas interativos descritos no primeiro ambiente [34, 62].

3.2.1 Ambiente de design

O ambiente de design inclui ferramentas para a descrição de modelos de controle de diálogo e também para a especificação de componentes de apresentação de interfaces de usuário. A ferramenta construída para a primeira tarefa é o editor Smart de diagramas Xchart. A segunda tarefa é apoiada pela ferramenta Grace.

A ferramenta Grace, composta por um editor de componentes de apresentação e um compilador responsável pela criação do código C++ equivalente, permite a definição da

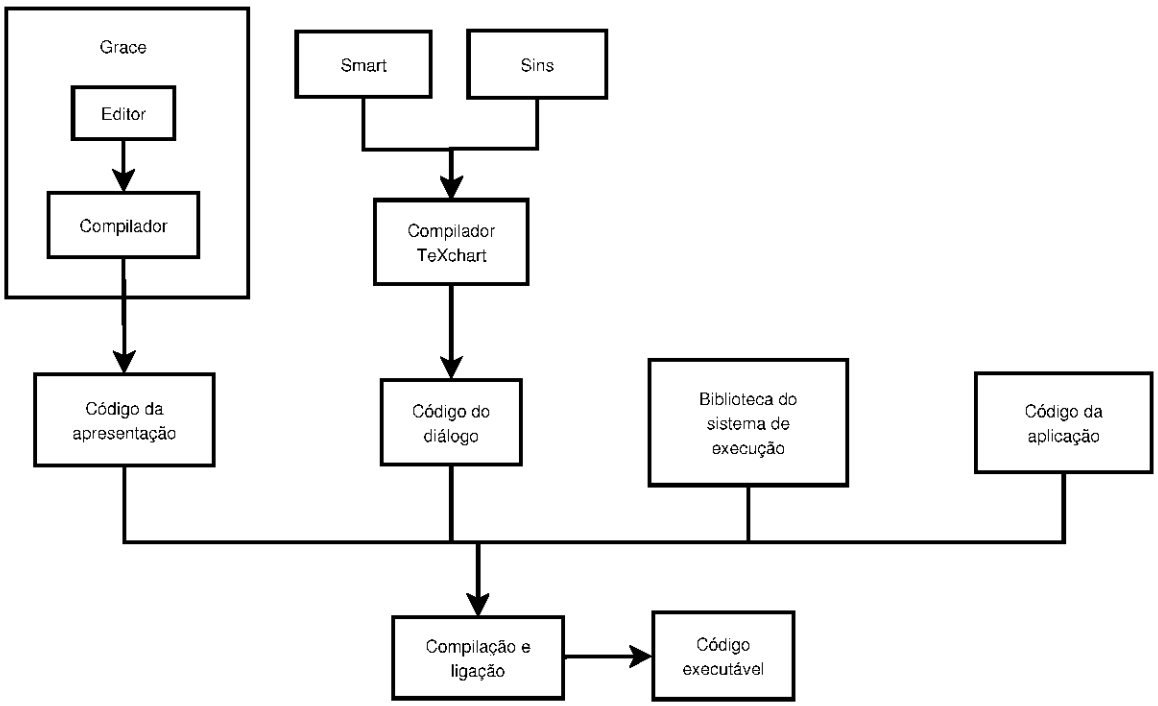


Figura 3.1: Ambiente de design da tecnologia Xchart

aparência da apresentação de uma interface de usuário através de um *toolkit* de geração de apresentações.

O sistema Smart permite a descrição de modelos na linguagem Xchart e é composto por um editor de diagramas Xchart, um tradutor que converte os diagramas criados para a linguagem textual TEXchart e um compilador TEXchart, que gera o código do componente de controle da interface no formato FSDX (Formato de Sistema Descrito em Xchart) [63]. Essa tradução ocorre de forma transparente para o projetista, que pode descrever os modelos em linguagem visual, mais apropriada à tarefa.

O código gerado é compilado e ligado à aplicação desenvolvida pelo projetista e à biblioteca do ambiente de execução. Este interage com a aplicação através do protocolo IPX (Interface de Programação de Xchart). No início da execução do sistema, o código gerado pelo compilador TEXchart, no formato FSDX, é carregado pelo ambiente de execução, que cria uma instância do modelo criado pelo projetista.

3.2.2 Sistema de execução

O sistema de execução (SE) tem por objetivo interpretar os modelos de controle de diálogo criados pelo projetista com o auxílio do ambiente de design. É composto por um servidor Xchart (SX), que interpreta localmente as especificações Xchart, e um gerente

de distribuição (GD), responsável pelo controle da execução distribuída do sistema reativo. As aplicações, assim como outros componentes de software que podem se comunicar com o sistema de execução, como a camada de apresentação da interface de usuário, são conhecidas como clientes.

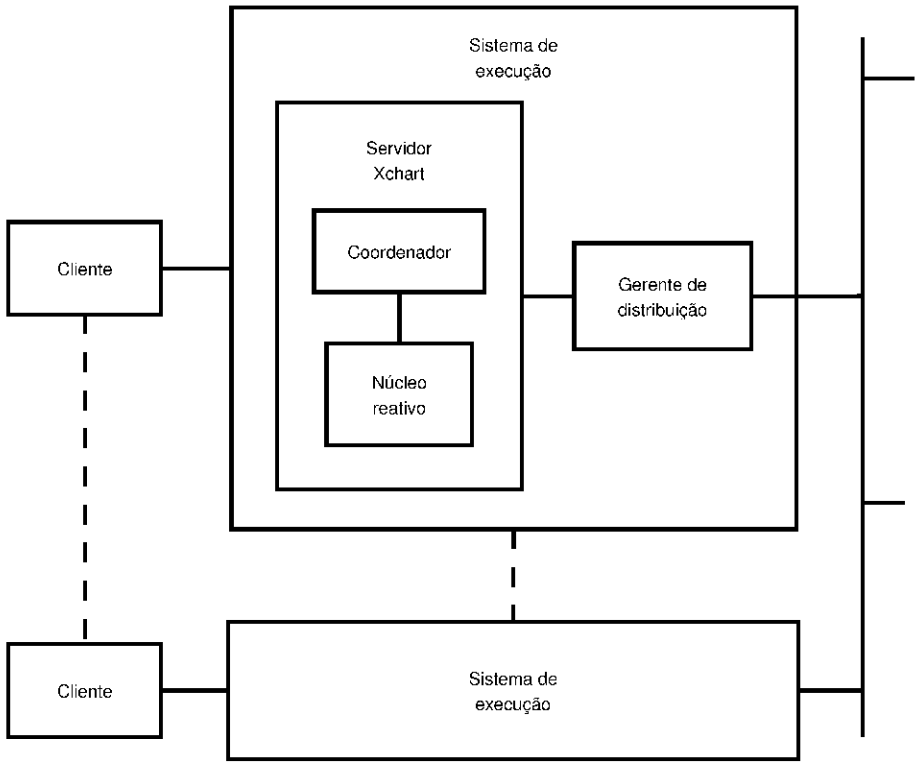


Figura 3.2: Sistema de execução da tecnologia Xchart

Durante a execução o modelo do controle de diálogo é carregado pelo sistema de execução e uma instância do modelo é criada. Os clientes se conectam ao sistema de execução através do protocolo IPX. Eventos são sinalizados pelos clientes com o envio de estímulos externos à fila de entrada da instância. O sistema de execução calcula a reação do modelo, como especificado pelo projetista, e deposita a reação nas portas de comunicação correspondentes.

3.2.2.1 Servidor Xchart

O servidor Xchart é responsável pela interpretação local dos modelos Xchart instanciados. A partir de um estímulo externo enviado por um dos clientes conectados, calcula a reação da instância correspondente, se esse cálculo puder ser feito localmente, ou repassa o estímulo ao gerente de distribuição, que propagará o sinal aos gerenciadores correspondentes às instâncias envolvidas. A reação, calculada localmente ou recebida de um

dos gerenciadores de distribuição contactados, é depositada na porta de comunicação da instância.

Para realizar suas tarefas, o servidor Xchart é composto por dois componentes: um coordenador e um núcleo reativo. O coordenador é responsável pelo escalonamento da execução das instâncias dos modelos Xchart, assim como a alocação e atualização de estruturas de memória internas e o acesso ao relógio do sistema e a outros recursos do ambiente operacional local. O núcleo reativo implementa a maior parte da especificação formal da linguagem Xchart. É responsável por depositar a reação da instância a um estímulo externo a partir da análise da configuração da instância. O núcleo reativo é o único componente do sistema que pode atualizar o estado de uma instância.

3.2.2.2 Gerente de distribuição

O gerente de distribuição mantém a consistência dos dados globais do sistema e gerencia o envio e recebimento de eventos entre os potencialmente vários sistemas, distribuídos, por exemplo, ao longo de uma rede local. É através do GD que as instâncias de modelos Xchart locais se comunicam, intermediadas pelos outros componentes do sistema, com instâncias em outros sistemas. Quando a comunicação remota é necessária, o coordenador do SX local delega essa tarefa ao GD local, que se comunica com os gerentes de distribuição remotos.

3.3 Linguagem Xchart

A linguagem Xchart, assim como a linguagem Statechart, na qual a primeira foi baseada, é uma extensão de diagramas de transição de estados. Assim como suas antecessoras, Xchart é visual: o projetista constrói diagramas que modelam o comportamento do sistema. Diagramas Xchart descrevem o comportamento de um sistema reativo que transita de uma configuração de estados correntes à seguinte, orientado por regras de transição ativadas por eventos.

Uma especificação Xchart [13] contém variáveis de controle globais (seção 3.3.1.1), uma hierarquia de eventos (seção 3.3.2), uma hierarquia de estados (seção 3.3.3) e transições (seção 3.3.4) que relacionam estados entre si. Outros elementos também podem participar da especificação e são descritos nas próximas seções.

3.3.1 Variáveis de controle

Variáveis de controle são repositórios que armazenam valores inteiros ou lógicos. Seus valores podem ser acessados e modificados por elementos de instâncias como gatilhos e regras, e também por clientes acessando variáveis globais através do servidor Xchart.

Variáveis são declaradas textualmente, em locais definidos dentro dos diagramas. Cada declaração inicia com a palavra-chave *INT* (para variáveis inteiras) ou *BOOL* (para variáveis lógicas), seguida dos nomes das variáveis declaradas separados por vírgulas, terminando por um símbolo de ponto-e-vírgula (;). Imediatamente após o nome de cada variável, pode-se acrescentar um símbolo de igual (=) e o valor inicial da variável (um valor inteiro para variáveis inteiras ou um valor lógico dentre *TRUE* e *FALSE* para variáveis lógicas). Caso nenhum valor inicial seja definido, variáveis inteiras iniciam com valor 0 e variáveis lógicas com valor *TRUE*.

3.3.1.1 Globais

Variáveis globais são compartilhadas por todas as instâncias Xchart, que podem acessar e modificar seus valores. São declaradas em um elemento especial da linguagem Xchart, posicionado fora de todos os diagramas, como mostra a figura 3.3.

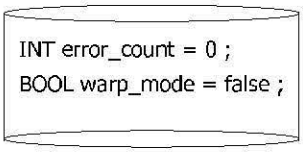


Figura 3.3: Declaração de variáveis de controle globais

3.3.1.2 Locais

Variáveis locais são exclusivas de cada instância Xchart, não sendo compartilhadas mesmo entre instâncias do mesmo diagrama. São declaradas textualmente, dentro do diagrama à qual pertencem, como no exemplo da figura 3.4.

3.3.2 Hierarquia de eventos

A linguagem diferencia entre dois tipos de eventos: instâncias de tipos de eventos primitivos e eventos propriamente ditos. Instâncias de tipos de eventos primitivos, ou apenas eventos primitivos, são abstrações usadas para representar "...acontecimentos instantâneos, não persistentes, ocorridos em um instante particular" e são criados externamente ou por instâncias Xchart. Tipos de eventos primitivos podem ser organizados numa hierarquia, que é descrita no diagrama Xchart e se mantêm constante durante a existência da instância do diagrama. Um gatilho que é ativado por um evento e também é ativado por qualquer ascendente de e na hierarquia de eventos.

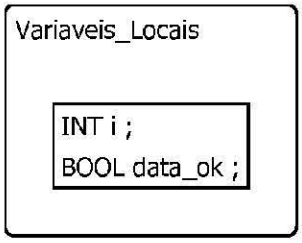


Figura 3.4: Declaração de variáveis de controle locais

Uma hierarquia de eventos é representada como uma árvore cuja raiz é o evento mais primitivo, como exemplificado na figura 3.5. Cada nó da árvore, correspondente a cada tipo de evento primitivo, é representado como um retângulo contendo o nome do tipo.

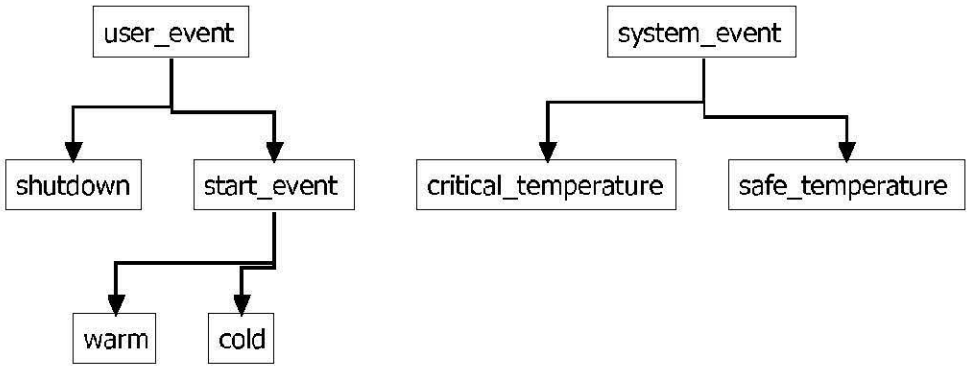


Figura 3.5: Exemplo de hierarquia de tipos de eventos primitivos

3.3.3 Hierarquia de estados

3.3.3.1 Estados

Segundo a especificação da linguagem Xchart [13], um estado "...captura uma situação conceitual, contexto ou modo". Todo estado de uma instância Xchart contém um atributo: ele pode estar ativo ou inativo, durante a execução do sistema. Cada estado pode ter um identificador, necessário para referência em alguns dos elementos do diagrama.

Estados podem ser refinados, contendo subestados que descrevem partes do comportamento do estado pai. Essas relações hierárquicas induzem uma árvore de estados, onde os filhos de um estado na árvore correspondem aos seus subestados e a raiz é um diagrama Xchart (o estado que contém todos os outros estados da hierarquia). A ativação de um estado implica na ativação dos seus subestados, de acordo com o tipo de refinamento usado. Da mesma forma, sua desativação leva à desativação dos seus subestados. Por outro lado, a desativação de um subestado por uma transição a um estado que não é descendente do estado pai leva à desativação do estado pai.

Estados são representados por retângulos de bordas arredondadas, com o identificador do estado, se houver, representado no interior do retângulo. Os subestados de estados compostos são representados totalmente contidos no interior do seu estado pai. A figura 3.6 apresenta exemplos de estados compostos, com refinamento exclusivo e concorrente.

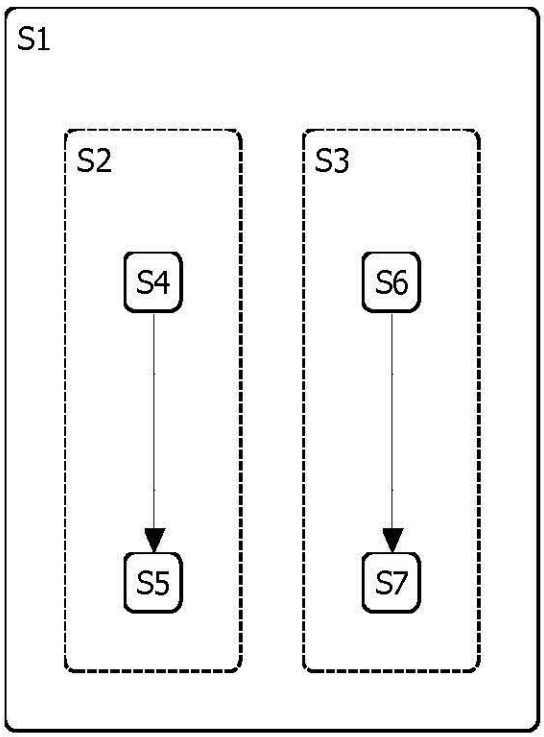


Figura 3.6: Estados compostos com refinamento exclusivo e concorrente

3.3.3.2 Tipos de refinamento

A linguagem Xchart admite dois tipos de refinamento de estados compostos: exclusivo e concorrente.

Estados exclusivos podem ser decompostos em subestados que se tornam ativos exclusivamente: apenas um dos subestados pode estar ativo a cada passo. São permitidas transições entre subestados de um mesmo estado exclusivo, e também entre esses subestados e estados não contidos no estado pai.

Estados concorrentes, cujos subestados são representados por retângulos de bordas tracejadas, contêm subestados que sempre estão ativos quando o estado pai se torna ativo: todos os subestados estão ativos a cada passo, desde que o estado pai esteja ativo. Não são permitidas transições entre subestados de estados concorrentes, mas são possíveis transições entre esses subestados e estados externos ao estado pai.

3.3.3.3 Segmentação

A linguagem Xchart prevê a segmentação de um diagrama em setores desconexos, à critério do projetista, relacionados entre si através da presença de estados de mesmo nome em mais de um setor. Dessa forma, um estado pode ser representado de forma abstrata em um setor de um diagrama, sendo descrito mais detalhadamente, inclusive sua decomposição em subestados, em outro setor da especificação.

Um estado que é descrito mais detalhadamente em outro setor do diagrama contém um triângulo invertido dentro do retângulo que o representa na versão parcialmente especificada. Caso o estado esteja representado de forma parcial em outros setores do diagrama, o retângulo que contêm a representação detalhada do estado inclui um triângulo no sentido normal. Ambas as situações são exemplificadas na figura 3.7.

3.3.4 Transições

Transições relacionam estados de um diagrama Xchart, sendo a única forma de ativar ou desativar estados. São compostas pelo conjunto de estados de origem (ou um estado fictício), um rótulo (formado por um gatilho e uma ação) e um conjunto de estados de destino/símbolos de história (ou um estado fictício).

A cada passo da execução de uma instância de diagrama, as transições cujos estados de origem estão ativos e regras estão habilitadas são executadas. Isso significa que os estados de origem são desativados (somente se a transição é externa), a ação definida no rótulo da transição é executada e os estados de destino são ativados.

Se a origem for um estado fictício inicial, representado por um círculo, os estados de destino serão ativados quando o estado que os contêm for ativado. Se o destino for um

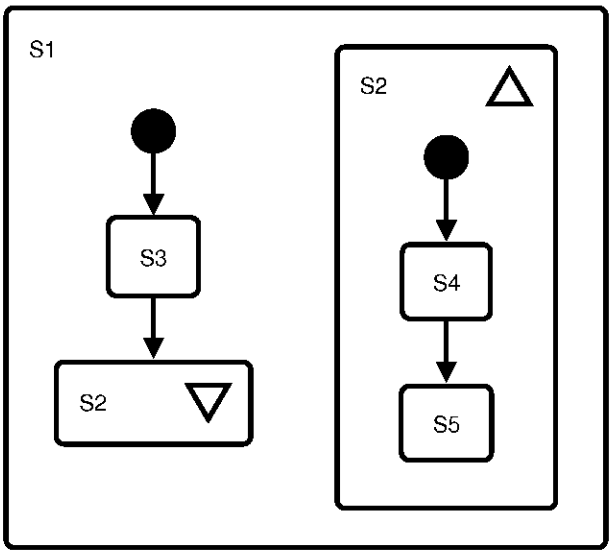


Figura 3.7: Exemplo de segmentação de estados

estado fictício final, também representado por um círculo, a instância correspondente é finalizada. Estados finais sempre são localizados fora do estado mais externo da diagrama.

Uma transição é representada por curvas que conectam os estados de origem e de destino. O rótulo da transição, correspondente à regra associada a ela, é representado textualmente. Se o rótulo for vazio, a transição é seguida como resposta a qualquer evento. As curvas tocam os estados de destino com pontas na forma de triângulos preenchidos. Transições internas são representadas totalmente contidas no estado de origem. Transições externas tocam estados de destino localizados no exterior do estado de origem. Transições são exemplificadas na figura 3.8.

3.3.4.1 História

Se o destino for um estado fictício de história, representado pelo nome do símbolo de história inscrito numa circunferência, a função de história é ativada. O estado fictício de história deve estar contido em um estado composto exclusivo. Quando a transição para o símbolo de história for executada, a ativação dos subestados não ocorre orientada pelos estados iniciais: em vez disso, o subestado que estava ativo pela última vez é o escolhido para ser ativado novamente. Caso seja a primeiro uso da função história nesse estado, o subestado a ser ativado é escolhido normalmente, através da orientação dada por estados iniciais. A função de história estrela, representada pelo nome do símbolo de história concatenado com um asterisco (*) inscrito numa circunferência, é equivalente à função de história, mas a ativação dos subestados contidos nos subestados também ocorre pela função história, recursivamente.

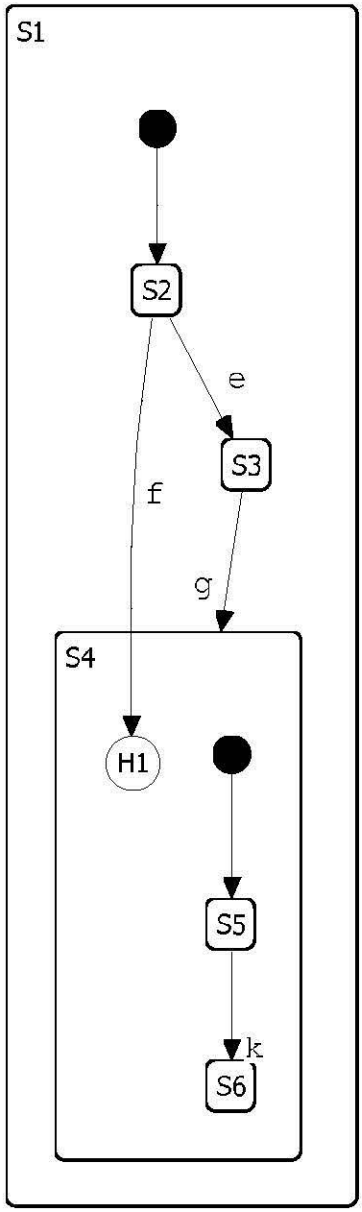


Figura 3.8: Exemplo de transições em diagrama Xchart

3.3.4.2 Prioridade

Duas transições são consideradas excludentes se ambas estiverem habilitadas num mesmo micro-passo e a execução de uma delas impedir a execução da outra. Isso acontece quando, no mesmo micro-passo, duas transições levariam à desativação de um mesmo estado duas vezes ou à ativação simultânea de mais de um subestado de um estado composto exclusivo. Nesses casos, a linguagem Xchart define os seguintes fatores de desempate, usados para escolher apenas uma de um conjunto de transições excludentes:

- seleção da transição de maior prioridade
  É possível associar um atributo de prioridade às transições. Esse atributo é um valor inteiro: quanto maior o valor, maior a prioridade. Caso nenhuma prioridade seja definida, o valor 0 é assumido. A prioridade de uma transição é representada como um número inscrito numa circunferência, posicionado próximo à curva que representa a transição. Um exemplo é dado na figura 3.9.

- seleção da transição mais interna
  Caso as transições tenham a mesma prioridade, a mais interna é selecionada para execução. Uma transição é considerada mais interna se seu ACMP (Ancestral Comum Mais Próximo–o estado que contém o mínimo de subestados e contém todos os estados associados à transição) é descendente do ACMP da outra transição.

- seleção arbitrária de uma transição
  Se nenhuma das transições puder ser escolhida através dos critérios anteriores, uma delas é selecionada arbitrariamente.

3.3.5 Regras

Uma regra é a união de um gatilho e uma ação. Regras são definidas textualmente, no formato *g: a*, sendo *g* o gatilho e *a* a ação, ou simplesmente *:a*, quando o gatilho é nulo, ou apenas *g*, quando a ação é nula. São associadas a transições ou a estados: caso a associação com uma transição não seja explícita, assume-se que uma referência a uma regra trata de uma regra associada a estado.

A cada passo, as regras cujo gatilho está habilitado são selecionadas e suas ações executadas. Regras associadas a transições também levam à desativação dos estados de origem e à ativação dos estados de destino. Quando associada a um estado, uma regra só é executada se o estado está ativo. Uma regra é executada até uma vez por passo, para evitar problemas causados por circularidade na habilitação.

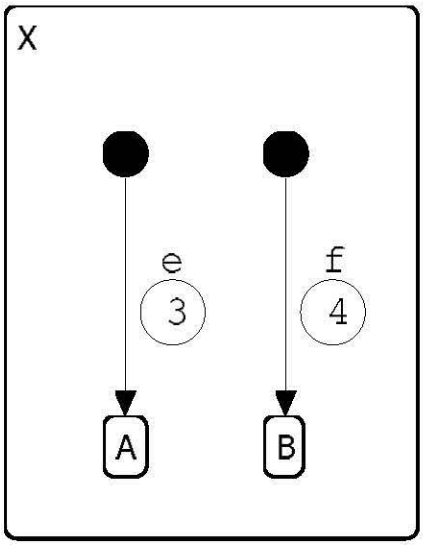


Figura 3.9: Exemplo de prioridades em transições excludentes

Quando associadas a transições, regras são representadas textualmente, próximas da curva que liga os estados de origem e destino, como descrito na seção 3.3.4. Associadas a estados, são representadas também textualmente, no interior do estado correspondente.

3.3.6 Gatilhos

Gatilhos, representados textualmente no formato *e[c]*, são compostos por uma lista de eventos *e* e uma condição *c*. Um gatilho deve estar habilitado para que a transição ou regra que o contém seja executada. Um gatilho é considerado habilitado quando os eventos *e* ocorrem e a condição *c* é verdadeira.

A lista de eventos é composta pelo nome de tipos de eventos primitivos. Se mais de um evento por especificado, estes devem ser unidos pela palavra-chave *AND*, para que a ocorrência de todos os eventos simultaneamente seja necessária, ou pela palavra-chave *OR*, para que a ocorrência de qualquer dos eventos especificados garanta a habilitação do gatilho. Um evento precedido pela palavra *NOT* leva à habilitação do gatilho quando ocorre qualquer evento que não o especificado.

Tanto o evento quanto a condição são opcionais: se nenhum evento for definido, um evento nulo é assumido e qualquer evento pode habilitar o gatilho; se nenhuma condição for especificada, a condição *[1=1]* (sempre verdadeira) é assumida.

3.3.6.1 Gatilhos especiais

A linguagem oferece os seguintes gatilhos especiais para serem usados em lugar do tipo de evento primitivo nos gatilhos de regras:

- *entry*
  O gatilho é considerado habilitado se a condição for verdadeira e o estado que o contém se tornará ativo no micro-passo corrente. É o gatilho padrão em gatilhos de regras associadas a estados.

- *exit*
  Equivalente ao gatilho entry, mas é habilitado quando o estado que o contém se torna inativo.

3.3.6.2 Funções de eventos

As seguintes funções podem ser usadas na construção da lista de eventos do gatilho:

- *en(e)*
  Evento gerado quando o estado *e* se torna ativo.

- *ex(e)*
  Evento gerado quando o estado *e* é desativado.

- *ch(expr)*
  Evento gerado quando a expressão *expr* tem seu valor alterado.

- *fs(expr)*
  Evento gerado quando a expressão *expr* se torna falsa.

- *tr(expr)*
  Evento gerado quando a expressão *expr* se torna verdadeira.

- *every(expr)*
  Evento gerado repetidamente, com período em segundos dado pela expressão *expr*.

- *after(expr)*
  Evento gerado após o número de segundos dado pela expressão *expr* terem se passado.

- *at(h:m:s)*
  Evento gerado quando o relógio do sistema chega à hora dada por *h:m:s*, onde *h* é a hora, *m* o minuto e *s* o segundo.

- *started(a)*
  Evento gerado quando a atividade a é iniciada.

- *stopped(a)*
  Evento gerado quando a atividade a é encerrada.

3.3.7 Expressões

3.3.7.1 Constantes

- *MAXINT*
  Corresponde ao maior valor que as variáveis de controle inteiras podem armazenar.

- *MININT*
  Corresponde ao menor valor que as variáveis de controle inteiras podem armazenar.

- *TRUE*
  Corresponde ao valor lógico verdadeiro.

- *FALSE*
  Corresponde ao valor lógico falso.

3.3.7.2 Operadores

Os operadores a seguir recebem dois valores inteiros e retornam um novo valor inteiro: + (soma), - (subtração), * (multiplicação) e / (divisão).

Os operadores a seguir recebem dois valores inteiros e retornam um novo valor lógico: < (menor), <= (menor ou igual), > (maior), >= (maior ou igual), = (igual) e <> (diferente).

Os operadores a seguir recebem dois valores lógicos e retornam um novo valor lógico: *OR* (disjunção), *AND* (conjunção) e *NOT* (negação).

3.3.7.3 Funções

As funções a seguir retornam um valor lógico:

- *in(e)*
  Retorna verdadeiro se o estado e está ativo.

- *ny(e)*
  Retorna verdadeiro se a lista de eventos e não foi responsável pela execução do passo corrente nem foi gerado durante o tratamento desse passo.

- *active(a)*
  Retorna verdadeiro se a atividade a está em execução.

- *hanging(a)*
  Retorna verdadeiro se a atividade a está suspensa, depois de ter a execução iniciada.

3.3.7.4 Step

O valor das variáveis de controle pode ser modificado várias vezes ao longo de um passo, que é composto de micro-passos. A função step pode ser usada para obter o valor que uma variável armazenava no início de um passo.

- *step(v)*
  Retorna o valor que a variável v armazenava no início do passo corrente da instância.

3.3.8 Ações

Ações são a reação da instância aos estímulos externos recebidos. São executadas com a habilitação dos gatilhos associados à transições ou regras. Ações podem ser combinadas para descrever comportamentos mais complexos. Existem ações para atribuição de valores a variáveis, para gerar ou esperar por eventos, gerenciar atividades, criar instâncias de diagramas Xchart, limpar a história de ativação de estados. Ações podem ser definidas como atômicas e conter construções de repetição e decisão. Tanto os valores correntes das variáveis de controle quanto os valores que estas tinham no início do passo podem ser acessados.

Ações são representadas textualmente, fazendo parte do rótulo de transições ou regras. Nas próximas seções são listados os tipos de ações disponíveis na linguagem:

3.3.8.1 Atribuição

- $v := e$
  Atribui o valor da expressão e à variável de controle v. O tipo do valor retornado pela expressão deve ser igual ao tipo da variável (inteiro ou lógico).

3.3.8.2 Geração de eventos

- *raise(e)*
  Gera um evento primitivo do tipo e na instância em que a ação é executada.

- *raise(e, xchart)*
  Gera um evento do tipo e em todas as instâncias do diagrama *xchart*.

- *raise(e, all)*
  Gera um evento do tipo e em todas as instâncias do sistema.

- *raise(e, id, estado)*
  Gera um evento do tipo e no estado *estado* da instância *id*.

3.3.8.3 Espera por eventos

- *wait(e)*
  Bloqueia a execução da instância corrente até que o evento e ocorra.

- *wait(e, k)*
  Bloqueia a execução da instância até que o evento e ocorra ou que k unidades de tempo se passem.

3.3.8.4 Gerência de atividades

Uma atividade é um processo executado por um cliente. Atividades são iniciadas e finalizadas através da comunicação entre as instâncias e os clientes, realizada através de portas de comunicação.

- *sync(a)*
  Inicia a execução da atividade a e bloqueia a instância corrente até que a atividade termine.

- *start(a)*
  Inicia a execução da atividade a mas mantém a instância corrente ativa.

- *stop(a)*
  Finaliza a execução da atividade a.

- *stop(all)*
  Finaliza a execução de todas as atividades.

- *suspend(a)*
  Suspende a execução da atividade a.

- *resume(a)*
  Retoma a execução da atividade suspensa a.

3.3.8.5 Criação de instâncias

- *call(xchart, id)*
  Cria uma instância do diagrama *xchart*, associado ao identificador *id*, e bloqueia a instância onde foi executada a ação até a finalização da instância criada.

- *run(xchart, id)*
  Cria uma instância do diagrama *xchart*, associado ao identificador *id*. A instância corrente não é bloqueada: ambas as instâncias executam concorrentemente.

3.3.8.6 Limpeza de história

Símbolos de história, descritos na seção 3.3.4, contêm um atributo de último estado que inicialmente é vazio. Quando um subestado do estado que contém o símbolo de história se torna ativo, esse subestado é armazenado no atributo de último estado do símbolo de história.

- *clear(h)*
  Limpa, no símbolo de história h, o atributo de último estado.

3.3.8.7 Atomicidade

Ações podem ser compostas, sendo separadas por ponto-e-vírgula (;), fazendo com que as ações primitivas definidas sejam executadas seqüencialmente. Ações compostas, de repetição e de decisão são executadas em mais de um micro-passo (um passo da execução da instância é composto de um ou mais micro-passos), sendo por isso consideradas divisíveis; todas as outras ações são indivisíveis. Quando é necessário garantir a atomicidade de ações divisíveis, que podem ser executadas concorrentemente à outras ações, a construção *atomic* deve ser usada.

- *atomic* $\{a_1; a_2; ...; a_n\}$;
  Executa a seqüência de ações $a_1; a_2; ...; a_n$ como uma ação atômica.

3.3.8.8 Repetição

- *while (cond)* $\{a_1; a_2; ...; a_n\}$;
  Executa a seqüência $a_1; a_2; ...; a_n$ enquanto o valor da expressão *cond* for igual a *TRUE*.

- *do* $\{a_1; a_2; ...; a_n\}$ *while (cond);*
  Executa a seqüência $a_1; a_2; ...; a_n$ pelo menos uma vez e repete a execução enquanto o valor da expressão *cond* for igual a *TRUE*.

- *for (atr1; cond; atr2) {a_1; a_2; ...; a_n};*
  Executa a atribuição *atr1* e, enquanto o valor da expressão *cond* for igual a *TRUE*, executa a seqüência a_1; a_2; ...; a_n e em seguida a atribuição *atr2*.

3.3.8.9 Decisão

- *if (cond) {a_1; a_2; ...; a_n} else {b_1; b_2; ...; b_n}*
  Se o valor da expressão cond for igual a TRUE, executa a seqüência a_1; a_2; ...; a_n; caso contrário, executa a seqüência b_1; b_2; ...; b_n.

3.4 Linguagem TEXchart

A linguagem TEXchart é a equivalente textual da linguagem visual Xchart. Um arquivo TEXchart oferece o mesmo poder expressivo da linguagem Xchart, permitindo a especificação de mais de um diagrama. Esse arquivo é compilado pelo compilador TEXchart, produzindo uma saída que pode ser usada para instanciar o diagrama pelo Sistema de Execução (ver seção 3.2.2 para mais detalhes). O arquivo pode ser gerado, pelo compilador, nas linguagens C, C++ ou Java ou ainda no formato XML [62].

Um arquivo TEXchart é estruturado em seis seções principais. Portas, atividades, variáveis e eventos são declarados em seções no início do arquivo, seguidos pela descrição de cada diagrama na seção correspondente e por uma seção para a documentação dos modelos descritos. Cada seção inicia com a palavra-chave que a define e um símbolo de abre chaves ({), terminando com um símbolo de fecha chaves (}). Seções podem ser aninhadas: nesse caso, cada símbolo de fecha chaves indica o fim da última seção aberta. Todas as seções do formato TEXchart, explicadas a seguir, são obrigatórias e devem aparecer exatamente na ordem aqui descrita.

3.4.1 Portas

A seção de portas contém a lista de identificadores das portas associadas aos diagramas descritos no arquivo. Inicia com a palavra-chave *PORTS*, contendo a lista de identificadores separados por vírgulas, como mostra a figura 3.10.

```
PORTS { user1, user2, environment }
```

Figura 3.10: Exemplo de seção de portas em arquivo TEXchart

3.4.2 Atividades

Esta seção contém a lista de identificadores das atividades que podem ser iniciadas pelas instâncias dos diagramas descritos. Inicia com a palavra-chave *ACTIVITIES* e contém a lista de identificadores separados por vírgulas, como mostra o exemplo da figura 3.11.

```
ACTIVITIES { process_data, send_data }
```

Figura 3.11: Exemplo de seção de atividades em arquivo TEXchart

3.4.3 Variáveis

Na seção de variáveis são declaradas as variáveis globais do sistema descrito no arquivo. Inicia com a palavra-chave *VARIABLES* e contém uma ou mais declarações de variáveis, como na linguagem Xchart. Um exemplo é mostrado na figura 3.12.

```
VARIABLES {
  INT error_count := 0;
  BOOL warp_mode := FALSE;
}
```

Figura 3.12: Exemplo de seção de variáveis globais em arquivo TEXchart

3.4.4 Eventos

A hierarquia de tipos de eventos primitivos é definida na seção de eventos. Essa seção inicia com a palavra-chave *EVENTS* e contém cada uma das árvores de tipos de eventos, separadas por vírgulas. Cada árvore é representada pelo identificador do tipo de evento raiz seguido pelas suas subárvores, separadas por vírgulas e contidas entre chaves. Dessa forma, subárvores são aninhadas nas suas árvores respectivas, descrevendo uma floresta que representa a hierarquia de tipos de eventos primitivos do sistema. A figura 3.13 mostra um exemplo de seção de eventos.

3.4.5 Xcharts

Cada diagrama Xchart é especificado em sua própria seção, que inicia com a palavra-chave *XCHART* seguida pelo identificador do diagrama. Dentro da seção de cada diagrama

```
EVENTTYPES {
  user_event {
    start { warm, cold },
    shutdown
  },
  system_event {
    critical_temperature, safe_temperature
  }
}
```

Figura 3.13: Exemplo de seção de eventos em arquivo TEXchart

são declarados, nessa ordem, em suas respectivas subseções: variáveis locais, subestados, pontos de histórico, transições, regras e prioridades. Um exemplo completo é mostrado na figura 3.14.

3.4.5.1 Variáveis locais

As variáveis locais do diagrama são declaradas nesta seção, que inicia pela palavra-chave *LOCALVARS* e contém as declarações das variáveis, de maneira similar à seção de variáveis globais.

3.4.5.2 Subestados

Os subestados do diagrama são descritos na seção de subestados. Essa seção inicia com a palavra-chave *SUBSTATES*, seguida pela palavra-chave *BAS*, quando o diagrama não contém nenhum subestado, ou pelo tipo do estado (*OR*, para estados concorrentes, ou *AND*, para estados exclusivos). A seguir, entre chaves, são listados os subestados, separados por vírgulas. A descrição de cada subestado inicia com seu nome, seguido do seu tipo e, no caso de um subestado composto, a lista dos seus subestados filhos, separados por vírgulas e entre chaves. Dessa forma, a árvore que representa a hierarquia de subestados do diagrama é especificada pelo aninhamento das descrições de subestados.

3.4.5.3 História

Os estados fictícios de história são listados na seção de história. Essa seção inicia com a palavra-chave *HISTORY* e contém as declarações de símbolos de história. Cada declaração inicia com o identificador do símbolo de história, seguido pelo tipo de história

```
XCHART S1 {
  LOCALVARS { }
  SUBSTATES OR {
    S23 OR { S4 , S5 , S6 }
  }
  HISTORY { H1 STANDARD S23; }
  TRANSITIONS {
    t0: FROM { S5 } TO { S6 } DO { } WHEN { };
    t1: FROM { S5 } TO { S4 } DO { } WHEN { };
    t2: FROM { DOT(S1) } TO { S5 } DO { } WHEN { };
    t3: FROM { S4 } TO { S6 } DO { } WHEN { e };
  }
  RULES {
    r0: S1 DO { STOP(process); } WHEN { f };
  }
  PRIORITIES { p0: t3 := 3; }
}
```

Figura 3.14: Exemplo de diagrama em arquivo TEXchart

(*STANDARD*, para história padrão, ou *STAR*, para história estrela) e o identificador do estado que contém a função de história, terminando com um ponto-e-vírgula.

3.4.5.4 Transições

Na seção de transições são descritas as transições do diagrama. Esta seção inicia com a palavra-chave *TRANSITIONS* e contém as declarações de cada transição. Uma declaração inicia com seu identificador, seguido pelo símbolo de dois pontos (*:*) e a palavra-chave *FROM*, a origem da transição contida entre chaves, a palavra-chave *TO*, o destino entre chaves, a palavra-chave *DO* seguida da lista de ações entre chaves (cada ação é representada textualmente, como na linguagem Xchart) e a palavra-chave *WHEN* seguida do gatilho da transição entre chaves (também representado como na linguagem Xchart), terminando com um símbolo de ponto-e-vírgula.

Tanto a origem quanto o destino da transição podem ser, respectivamente, estados fictícios de início ou final, ou uma lista de estados. No primeiro caso, o estado fictício é representado pela palavra-chave *DOT* seguida do identificador do estado que o contém entre parênteses. No segundo caso, a lista de estados é representada por uma seqüência de identificadores de estados, separados por vírgulas. Um identificador de estado é prefixado com o símbolo de acento circunflexo (^) quando a transição é interna. Se a origem for um

estado de início, este deve ser o único elemento da origem; o destino da transição pode conter qualquer combinação de estados comuns e estados finais.

3.4.5.5 Regras

As regras associadas à estados do diagrama são descritas nesta seção, que inicia com a palavra-chave *RULES* e contém as declarações das regras. Cada declaração inicia com o identificador da regra, seguida pelo símbolo de dois pontos (*:*), o identificador do estado à qual a regra está associada, o tipo da regra (*ENTRY*, para regras executadas na ativação do estado, ou *EXIT*, quando executadas na desativação), a palavra-chave *DO* seguida pela lista de ações entre chaves (representada como na linguagem Xchart) e a palavra-chave *WHEN* seguida pelo gatilho da regra (também representado como na linguagem Xchart), terminando com um símbolo de ponto-e-vírgula (*;*). Caso o tipo da regra seja *ENTRY*, esta palavra-chave pode ser omitida da declaração.

3.4.5.6 Prioridades

As prioridades das transições, quando for necessário especificá-las, são definidas nesta seção, que inicia com a palavra-chave *PRIORITIES* e contém as declarações das prioridades. Cada declaração inicia com seu identificador, seguido do símbolo de dois pontos, o identificador da transição à qual a prioridade está associada, um símbolo de atribuição (*:=*) e o valor da prioridade (um número inteiro).

3.4.6 Documentação

A documentação associada aos elementos do diagrama é contida nesta seção, que inicia com a palavra-chave *DOCUMENTATION*. A documentação de cada elemento é representada dentro de sua própria subseção, que inicia com o tipo do elemento documentado (por exemplo: *XCHART*, *ACTIVITIES*, *LOCALVARS*), seguido por um ponto (.) e o identificador do elemento. No caso de elementos contidos em um diagrama, este também deve ser referenciado; após o identificador do diagrama ao qual o elemento pertence é colocado um ponto seguido pelo identificador do elemento. Um exemplo de documentação é mostrado na figura 3.15.

3.5 Conclusão

Para possibilitar a especificação de sistemas reativos e em especial de componentes de controle de diálogo de interfaces de usuário, a tecnologia Xchart abrange um ambiente de apoio, utilizado para o projeto e execução de tais sistemas, além de uma linguagem

```
DOCUMENTATION {
  XCHART.S1 { Example Xchart diagram. }
  STATE.S1.On {
    This state is active when the radio is on.
  }
  STATE.S1.Off {
    This state is active when the radio is off or
    in hibernating mode.
  }
}
```

Figura 3.15: Exemplo de seção de documentação em arquivo TEXchart

visual e uma linguagem textual para a descrição desses sistemas. Ambas as linguagens têm poder de expressão equivalente: modelos especificados numa linguagem podem ser traduzidos para a outra, e vice-versa.

Para o projeto de sistemas utilizando essas linguagens, é necessário um ambiente adaptado às necessidades do projetista e que apóie adequadamente a edição de diagramas. Esse ambiente, incluindo seus requisitos, é descrito em detalhes no próximo capítulo.

Capítulo 4

Visão Geral da Solução

4.1 Introdução

Tendo sido contextualizado, nos capítulos anteriores, o problema de desenvolver um sistema reativo com o auxílio da tecnologia Xchart, é apresentada uma solução a este problema que consiste na criação de um novo editor de diagramas Xchart. Esse problema específico é analisado em maiores detalhes na seção 4.2, dando origem aos requisitos descritos na seção seguinte. A interface do editor é descrita em termos gerais na seção 4.4 e as operações disponíveis na seção 4.5. Devido às limitações inerentes à linguagem TEXchart, uma extensão à essa linguagem é apresentada na seção 4.6.

4.2 Análise do problema

Como visto nos capítulos anteriores, sistemas reativos são uma classe importante de sistemas computacionais. O desenvolvimento desse tipo de sistema beneficia-se fortemente do uso de técnicas, linguagens e ferramentas especializadas. Entre essas linguagens, Xchart se destaca por introduzir construções que facilitam a especificação de sistemas reativos complexos. Sendo uma linguagem visual, é importante que exista uma maneira de criar e modificar especificações Xchart de forma direta, visual, sem a necessidade de traduções entre linguagens intermediárias como a equivalente textual de Xchart conhecida como TEXchart.

Para atender a essa demanda foi criado em meados dos anos 90 o editor Smart. Esse editor permite a edição de diagramas Xchart de maneira indireta, através da especificação da árvore de estados do sistema e, separadamente, das transições existentes entre eles. A especificação pode ser visualizada na sintaxe Xchart padrão, com a aplicação de um algoritmo de layout automático de grafos adaptado, mas não pode ser editada diretamente

nessa forma de apresentação. Essa limitação torna a edição de especificações mais lenta e propensa a erros porque exige um nível de indireção nas ações e interpretações do usuário, mesmo que a distância entre a árvore de estados e a especificação Xchart em si não seja tão grande quanto entre uma especificação visual e uma textual. Além disso, a integração com outras ferramentas do ambiente Xchart não está prevista no Smart, dificultando a execução, simulação e depuração das especificações editadas. Outra limitação relaciona-se com o algoritmo de layout automático implementado, que não se preocupa em criar mapas semelhantes aos criados manualmente nem em manter o mapa mental do usuário durante a edição dos diagramas.

4.3 Requisitos do editor Sins

4.3.1 Requisitos funcionais

Dadas as limitações do editor Smart e as características do desenvolvimento de sistemas reativos, surge a necessidade de um editor de diagramas Xchart capaz de atender à essas demandas. Optou-se pela criação de um novo editor porque a modificação do editor Smart exigiria mudanças significativas na sua arquitetura e não traria os benefícios da integração com um ambiente completo de desenvolvimento.

A esse editor deu-se o nome Sins (*Sins Is Not Smart*, em referência ao editor Xchart já existente). Os principais requisitos desse editor correspondem às limitações do editor Smart como discutidas anteriormente:

- *edição direta de diagramas*
  Diferentemente do editor Smart, o editor Sins possibilita a modificação direta das especificações Xchart representadas na linguagem visual Xchart, sem a necessidade de passos indiretos como a modificação de árvores de estados. Métodos alternativos de edição, como o preenchimento de modelos (*templates*) textuais ou visuais foram descartados a princípio por não diminuírem a distância semântica entre a representação da especificação e o modelo mental que o usuário tem da especificação.

- *integração com outras ferramentas*
  Essa integração ocorre com duas categorias de ferramentas: aquelas do ambiente Xchart (como o compilador e o simulador) e aquelas comuns ao desenvolvimento de sistemas (como controle de versões de arquivos e gerenciamento de equipes). A principal forma de integração com o ambiente Xchart é a leitura e escrita de especificações na linguagem TEXchart. No futuro, ligações mais diretas às ferramentas do ambiente permitirão a simulação de especificações utilizando-se do editor como interface de controle.

A integração com a segunda categoria de ferramentas é possibilitada pela implementação do editor Sins como extensão de um ambiente integrado de desenvolvimento. Dessa forma, o editor apresenta-se como mais uma das ferramentas oferecidas pelo ambiente. Isso torna transparente, do ponto de vista do editor Sins, o acesso às outras ferramentas necessárias à criação de sistemas computacionais complexos.

- *layout automático*
  O editor Sins implementa um algoritmo de layout automático que procura maximizar a legibilidade dos mapas gerados e se aproximar dos mapas criados manualmente.

4.3.2 Requisitos não-funcionais

Além dos requisitos funcionais, descritos na seção 4.3.1, o editor Sins apresenta requisitos não-funcionais, isto é, exigências quanto às características de sua implementação. Esses requisitos têm origem nas limitações do editor Smart e na forma como o ambiente Xchart é desenvolvido atualmente. Tais requisitos incluem:

- *independência de plataforma*
  A independência de plataforma pode ser definida como a capacidade de um mesmo sistema computacional ser executado em mais de um ambiente operacional, com nenhuma ou poucas modificações. As principais técnicas existentes para a criação de sistemas com essas características incluem o uso de máquinas virtuais [64] (como as plataformas Java e *.NET*) e a compilação do sistema para mais de uma arquitetura, muitas vezes com o auxílio de um framework voltado a essa tarefa, como o Qt [65] ou o WxWindow [66].

- *licença compatível com código livre*
  Esse requisito se aplica tanto ao editor Sins quanto às plataformas e frameworks utilizadas em seu desenvolvimento. Assim como as outras ferramentas do ambiente Xchart, o editor Sins tem o seu código fonte disponibilizado livremente, de forma que possa ser estudado e estendido por terceiros. Conseqüentemente, a licença do ambiente integrado de desenvolvimento (*IDE – Integrated Development Environment*) ao qual se integra deve permitir extensões licenciadas na forma de software livre.

- *extensibilidade*
  Como o requisito anterior, a extensibilidade deve ser principalmente uma característica do IDE ao qual o Sins se integra, mas também deve ser considerada durante a implementação do editor em si. Arquiteturalmente, essa característica é alcançada

através da fatoração adequada dos componentes do sistema. A existência de mecanismos de extensão públicos e bem documentados que possibilitem a instalação de extensões em tempo de execução, na forma de plugins ou módulos, traz grandes vantagens na depuração desses componentes.

- *integração com ambiente de desenvolvimento*
  Ambientes integrados de desenvolvimento oferecem recursos úteis para os processos de implementação e depuração de sistemas, tais como modificação facilitada de código durante a depuração. Um dos requisitos do editor Sins é se integrar ao IDE de forma a possibilitar a utilização desses recursos na implementação de sistemas na linguagem Xchart. Os pontos de integração possíveis incluem o editor de código fonte, a notificação de erros e problemas, a compilação automatizada do diagrama desenhado e a simulação passo a passo do sistema.

- *usabilidade*
  O editor Sins deve apresentar maior usabilidade que o editor Smart. Para isso, o editor conta com recursos como a edição direta dos diagramas e o layout automático. Além disso, sua interface de usuário segue o padrão dos editores de diagramas com os quais o usuário pode ter tido contato anteriormente.

4.4 Interface de usuário

A interface do editor Sins é gráfica, misturando elementos dos paradigmas de manipulação direta e WIMP, descritos nas seções 2.6.1 e 2.6.2, e reaproveitando a estrutura comum à maioria dos editores de diagramas existentes. Existe uma região principal da interface dedicada à exibição da especificação editada. Essa área de edição é cercada por barras de ferramentas e menus que dão acesso às operações que manipulam a especificação.

A área de edição apresenta vários modos de edição que podem ser alternados pelo usuário: Xchart (onde a especificação é representada na linguagem Xchart), eventos (onde a hierarquia de eventos da especificação editada pode ser modificada), portas, atividades e TEXchart (onde o código fonte TEXchart da especificação pode ser editado). As modificações em um modo são aplicadas à especificação editada e, dessa forma, são visíveis nos outros modos. Tanto o modo Xchart quando o modo de eventos incluem uma barra de ferramentas correspondente para ativação das operações de edição de especificações. O modo TEXchart corresponde a um editor de textos padrão do ambiente computacional, incluindo as capacidade comuns a esses tipos de editores (edição de textos, realce de sintaxe, acesso à área de transferência, etc.). A visão geral da interface de usuário no modo Xchart pode ser vista na figura 4.1.

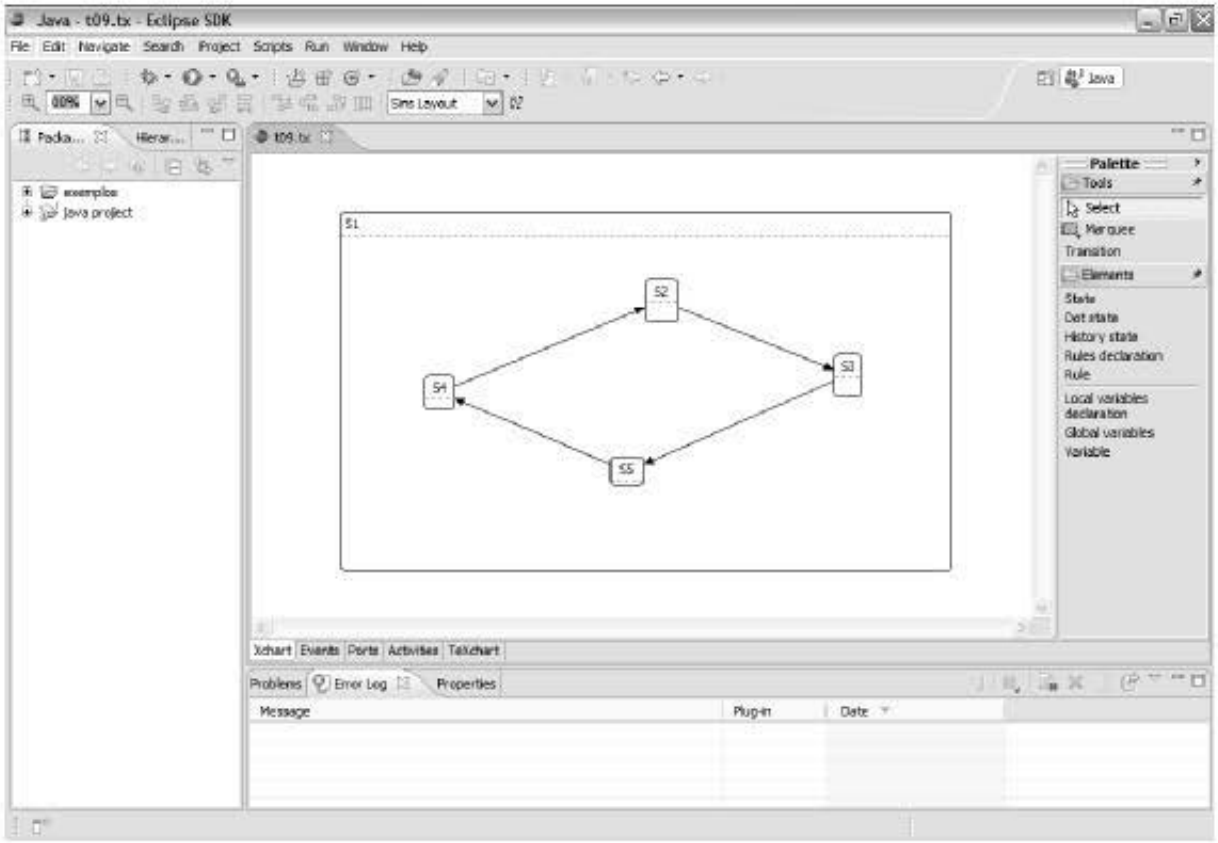


Figura 4.1: Visão geral da interface de usuário do editor Sins

Exemplos de diagramas editados nos modos de eventos, portas e TEXchart podem ser vistos nas figuras 4.2, 4.3 e 4.4, respectivamente. O modo de atividades é similar ao modo de portas.

Figura 4.2: Interface de usuário do modo de eventos

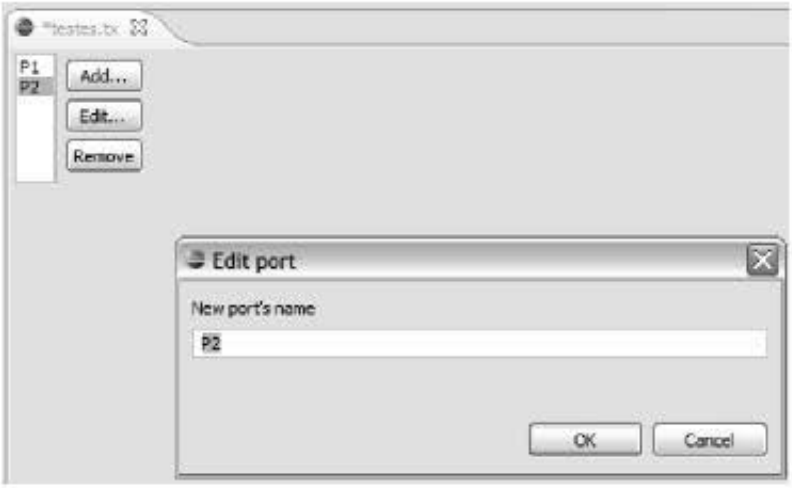


Figura 4.3: Interface de usuário do modo de portas

Erros de sintaxe no código fonte TEXchart da especificação são indicados pelo editor, como mostra a figura 4.5.

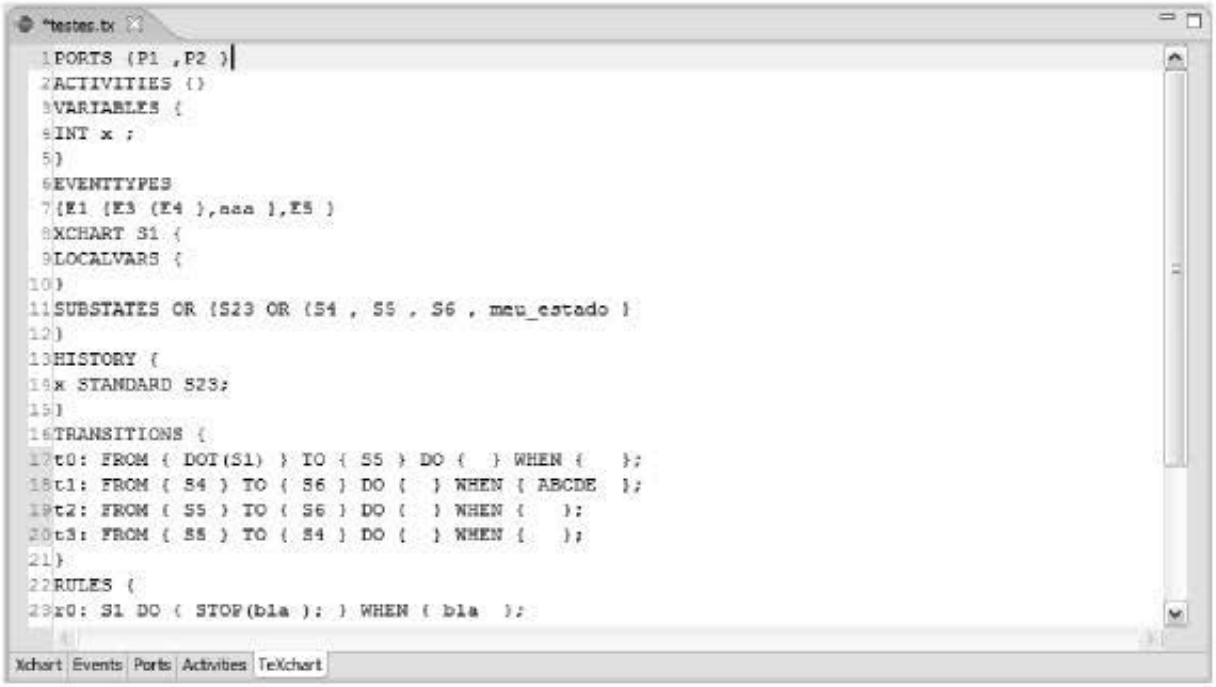


Figura 4.4: Interface de usuário do modo TEXchart

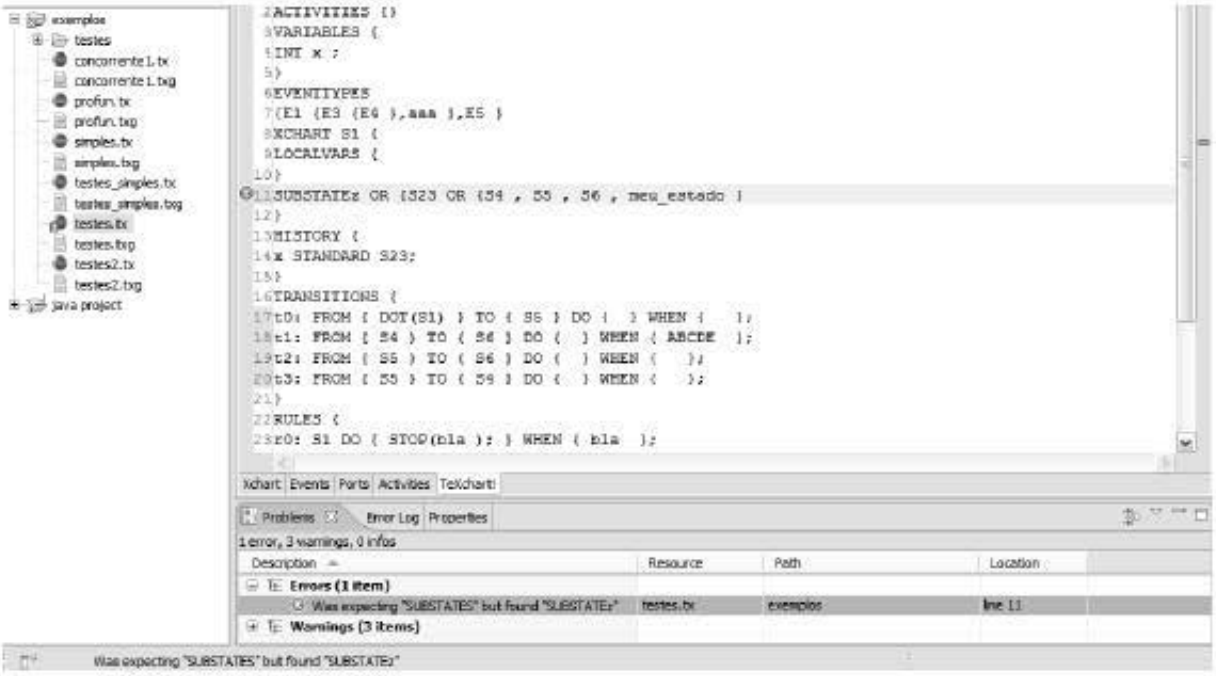


Figura 4.5: Sinalização de erros de sintaxe no modo TEXchart

4.4.1 Sintaxe das operações

As operações disponíveis são aplicadas através dos componentes da interface de usuário. Cada operação pode exigir parâmetros que são especificados antes da ou simultaneamente à ativação da operação, dependendo das suas características. As formas de aplicação de cada tipo de operação, descritas a seguir, seguem o padrão tradicionalmente usado em editores de diagramas, buscando facilitar a utilização do editor proposto pelo projetista.

4.4.1.1 Criação de elementos

As operações de criação de elementos são aplicadas com a execução de dois passos:

1. Seleção da operação, através do clique em um dos botões da barra de ferramentas
2. Indicação da posição inicial do novo elemento, que implica na ativação da operação

A indicação da posição do novo elemento pode ocorrer de duas formas:

- Um único clique no interior da área de edição.
- Uma operação de arrastar-e-soltar que inicia em uma das extremidades da região retangular a ser ocupada pelo novo elemento e termina na extremidade oposta.

4.4.1.2 Seleção de elementos

A seleção de elementos da especificação pode ocorrer de duas formas:

- Um clique no interior da representação do elemento.
- Uma operação de arrastar-e-soltar que inicia em uma das extremidades da região retangular ocupada pelos elementos a serem selecionados e termina na extremidade oposta.

Quando um elemento é selecionado, tornam-se visíveis os seus manipuladores. Manipuladores são quadrados preenchidos representados em pontos-chave do elemento, indicando que este está selecionado, e podem ser arrastados para modificar a sua forma. O número e a posição dos manipuladores depende do tipo de elemento selecionado: elementos hierárquicos apresentam 8 manipuladores (um em cada extremidade do retângulo correspondente e um no centro de cada lado do mesmo retângulo); transições apresentam 3 (um em cada extremidade e um no ponto intermediário). Os manipuladores de um estado podem ser vistos na figura 4.6.

Se apenas uma transição for selecionada, um símbolo de adição inscrito num quadrado próximo ao ponto intermediário da transição. Este símbolo, quando clicado, ativa a operação de adição de conexões à transição selecionada.

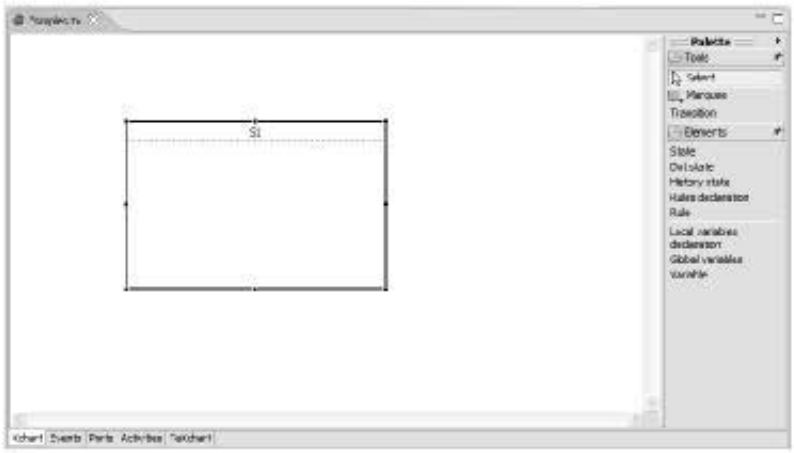


Figura 4.6: Manipuladores de um estado selecionado

4.1.1.3 Redimensionamento de elementos

O redimensionamento de elementos pressupõe a seleção de exatamente um elemento. Uma operação de arrastar-e-soltar sobre um de seus manipuladores resulta na modificação das dimensões do elemento, de forma a se adequar à nova posição dos manipuladores.

4.1.1.4 Movimentação de elementos

Um elemento pode ser movido através de uma operação de arrastar-e-soltar aplicada à sua representação na área de edição, resultando no seu reposicionamento relativo à distância percorrida na operação. Um elemento é automaticamente selecionado quando movido.

4.1.1.5 Exclusão de elementos

Um elemento selecionado é excluído com o pressionamento da tecla de atalho associada (*Delete*) ou através da ativação do item de menu correspondente.

4.4.1.6 Outras operações

Todas as outras operações são aplicadas com a ativação de um item de menu ou com o clique em um botão da barra de ferramentas, possivelmente precedida da seleção de elementos da especificação (dependendo da operação).

4.5 Operações

O editor oferece um conjunto de operações que podem ser aplicadas às especificações com o objetivo de atender às necessidades do projetista que utiliza esses serviços. São descritas operações para manipulação de especificações, edição de diagramas e modificação dos modos de edição Xchart e de eventos. Operações sobre o modo de edição TEXchart não são descritas em detalhes porque são padronizadas no ambiente ao qual o editor se integra.

4.5.1 Manipulação de especificações

Estas operações manipulam especificações como unidades inteiras.

4.5.1.1 Criar especificação vazia

Cria uma novo especificação, que será armazenada em um novo arquivo TEXchart. A especificação criada não contém nenhum elemento Xchart mas, seguindo as exigências da linguagem TEXchart, todas as seções principais que um arquivo nessa linguagem deve ter são inseridas no código fonte da especificação.

4.5.1.2 Abrir especificação existente

Abre uma especificação Xchart existente, armazenada num arquivo TEXchart. Se o arquivo selecionado não contiver informações de posição sobre os seus elementos, a operação de layout automático é aplicada; se informações de posição estiverem presentes no arquivo, estas são utilizadas no posicionamento dos elementos.

4.5.1.3 Salvar especificação

Salva a especificação que está aberta no editor em um arquivo TEXchart. A informação sobre a posição dos elementos do diagrama é escrita no arquivo de geometria correspondente.

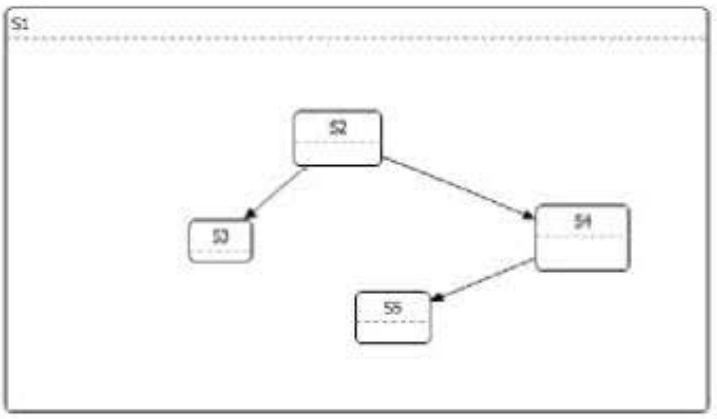
4.5.1.4 Imprimir diagramas

Imprime a visão da especificação correntemente ativa no editor.

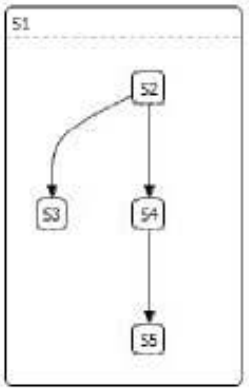
4.5.1.5 Aplicar layout automático

O editor permite o posicionamento e o dimensionamento livres dos elementos componentes dos diagramas Xchart. Além disso, os diagramas TEXchart existentes atualmente não con-

têm informação sobre posicionamento de elementos. Para obter diagramas compreensíveis e estéticos, usáveis, por exemplo, em documentos, é oferecida a funcionalidade de layout automático de diagramas, responsável por reposicionar os elementos destes sem interferir no comportamento reativo descrito. Um exemplo da aplicação de layout automático pode ser visto na figura 4.7.



(a) Antes



(b) Depois

Figura 4.7: Exemplo de aplicação de layout automático

4.5.2 Edição de diagramas

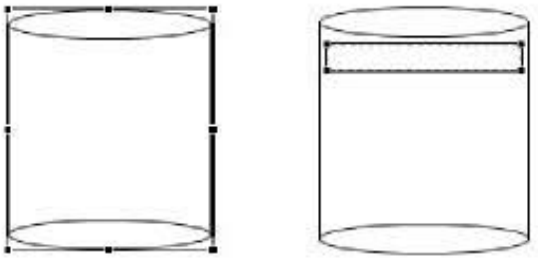
Estas operações são aplicáveis através das visões Xchart e de eventos, modificando a especificação até alcançar o objetivo do projetista usuário do editor.

4.5.2.1 Criar declaração de variáveis globais

Cria um símbolo para declaração de variáveis globais, como pede a linguagem Xchart. Dentro do símbolo, que pode ser posicionado em qualquer local da especificação não ocupado por outro elemento, há espaço para a inserção do texto das declarações. Apenas um símbolo de declaração de variáveis globais pode existir em cada especificação. Caso já exista uma declaração na especificação corrente, esta operação fica indisponível.

4.5.2.2 Modificar declaração de variáveis

Modifica as declarações de variáveis contidas num símbolo de declarações já existente na especificação Xchart corrente, permitindo a adição, remoção e edição de cada declaração individual. Essa operação é mostrada na figura 4.8.



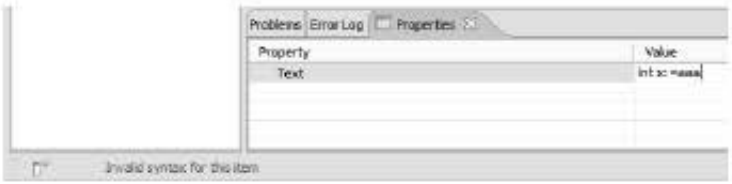
(a) Criação da declaração de variáveis (b) Criação de variável global

Figura 4.8: Exemplo de modificação de declaração de variáveis

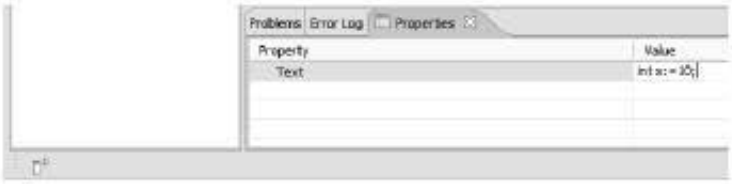
A edição de itens como declarações de variáveis e regras é restrita pela sintaxe definida para a linguagem Xchart. Edições com erros de sintaxe não são permitidas, como mostrado na figura 4.9. Na subfigura *a* uma declaração de variáveis está sendo preenchida com texto que tem erros de sintaxe. Esse erro é corrigido na subfigura *b*; confirmando a modificação, esta é aplicada ao diagrama, como mostra a subfigura *c*.

4.5.2.3 Criar novo estado

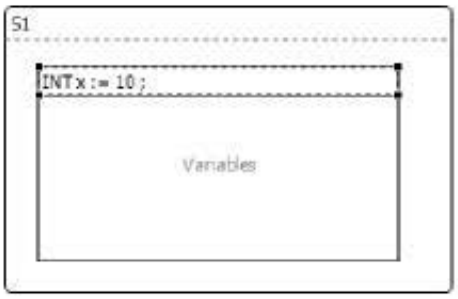
Cria um novo estado na especificação. O estado criado contém espaço para o nome que o identifica. Se o novo estado for posicionado fora de qualquer estado ou diagrama, é tratado como um novo diagrama.

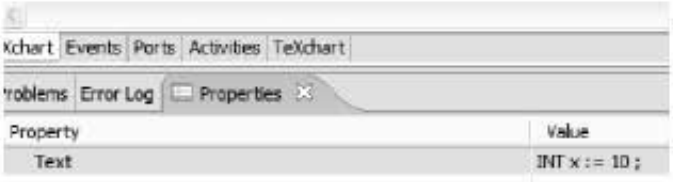


(a)



(b)





(c)

Figura 4.9: Tratamento de erros de sintaxe

4.5.2.4 Modificar tipo de estado composto

Modifica o tipo de um estado composto existente, alternando entre estados com decomposição exclusiva e concorrente.

4.5.2.5 Criar declaração de variáveis locais

Cria um símbolo para declaração de variáveis, posicionado dentro de um diagrama mas fora de qualquer outro subestado, oferecendo espaço para a inserção do texto das declarações. Cada diagrama pode conter no máximo um símbolo de declaração de variáveis locais.

4.5.2.6 Criar declaração de regras

Cria um símbolo para declaração de regras associadas a estados, posicionado dentro de um estado ou diagrama, oferecendo espaço para a inserção do texto das regras. Cada estado pode conter no máximo um símbolo de declaração de regras.

4.5.2.7 Modificar declaração de regras

Modifica o texto contido num símbolo de declaração de regras, permitindo a adição, remoção e edição de cada declaração individual. O novo texto das declarações deve seguir a sintaxe da linguagem Xchart.

4.5.2.8 Criar transição

Cria uma nova transição entre estados, que deve ter ambas as extremidades conectadas. A transição oferece espaço para o texto da regra associada, que inicialmente é vazia.

4.5.2.9 Modificar regra de transição

Modifica a regra associada à uma transição existente. O novo texto da regra deve seguir a sintaxe da linguagem Xchart.

4.5.2.10 Modificar prioridade de transição

Modifica a prioridade de uma transição existente. A nova prioridade deve ser um valor inteiro.

4.5.2.11 Modificar posições de transição

A posição de cada transição é dada pela combinação das posições de suas dobras e do seu rótulo. As dobras de uma transição são guiadas por pontos de referência que podem ser inseridos, movidos ou excluídos pelo usuário. O rótulo pode ser posicionado em qualquer lugar ao longo da transição.

4.5.2.12 Criar estado inicial/final

Cria um novo símbolo de estado inicial ou final, posicionado em qualquer região da especificação.

4.5.2.13 Criar símbolo de história

Cria um novo símbolo de história, posicionado no interior de qualquer estado. O símbolo oferece espaço para o nome que o identifica e inicialmente é do tipo padrão.

4.5.2.14 Modificar tipo de símbolo de história

Modifica o tipo de um símbolo de história existente, alternando entre história padrão e história estrela.

4.5.2.15 Criar novo tipo de evento

Cria um novo tipo de evento, para ser inserido na hierarquia de tipos de eventos de uma especificação Xchart. O novo tipo de evento pode ser posicionado em qualquer região livre da especificação.

4.5.2.16 Relacionar eventos

Cria uma nova relação de herança entre tipos de eventos. O evento de origem passa a ser o supertipo do evento de destino. Ciclos não são permitidos.

4.5.2.17 Modificar nome de elemento

Modifica o nome de um elemento existente; elementos cujo nome pode ser modificado incluem estados e símbolos de história. O formato do novo nome deve atender às exigências da linguagem Xchart. O novo nome deve ser diferente de todos os outros elementos de mesmo tipo contidos no diagrama.

4.5.2.18 Mover elemento

Modifica a posição de um elemento existente da especificação. Todos os elementos que podem ser criados pelas operações do editor podem ser movidos, dadas as restrições impostas pela linguagem Xchart.

Um estado pode conter subestados, declarações de variáveis locais e regras, mas esses elementos devem estar inteiramente contidos no estado pai. Um estado movido implica no reposicionamento de todos os elementos contidos no estado, inclusive seus pontos de conexão de transições.

Transições podem ser posicionadas em qualquer local, desde que estejam conectadas a um estado de origem e um estado de destino. Símbolos de declaração de variáveis globais devem ser posicionados fora de qualquer outro elemento. Símbolos de história devem estar contidos em um estado.

Com exceção de estados e transições, todos os tipos de elementos devem estar em locais livres, sem sobreposição a outros elementos.

4.5.2.19 Redimensionar elemento

Modifica as dimensões de um elemento Xchart existente na especificação, seguindo as mesmas restrições da operação mover elemento.

4.5.2.20 Excluir elemento

Exclui um elemento existente da especificação. Um estado excluído implica na exclusão de todos os elementos contidos nele. Eventos descendentes de um evento excluído são mantidos. Se o tipo de evento excluído for filho de outro tipo de evento, os eventos descendentes permanecem em uma árvore desconexa da árvore que continha o evento excluído.

4.5.3 Modos de edição Xchart e de eventos

Estas operações são aplicadas à visão Xchart, modificando-a com o objetivo de permitir a edição de partes diferentes do diagrama.

4.5.3.1 Aumentar/diminuir zoom

Esta operação aumenta ou diminui a escala da visão em intervalos pré-definidos.

4.5.3.2 Ajustar zoom

Ajusta a escala da visão de forma que a especificação inteira ocupe toda a área da visão. Um exemplo de um diagrama com zoom aplicado pode ser visto na figura 4.10.

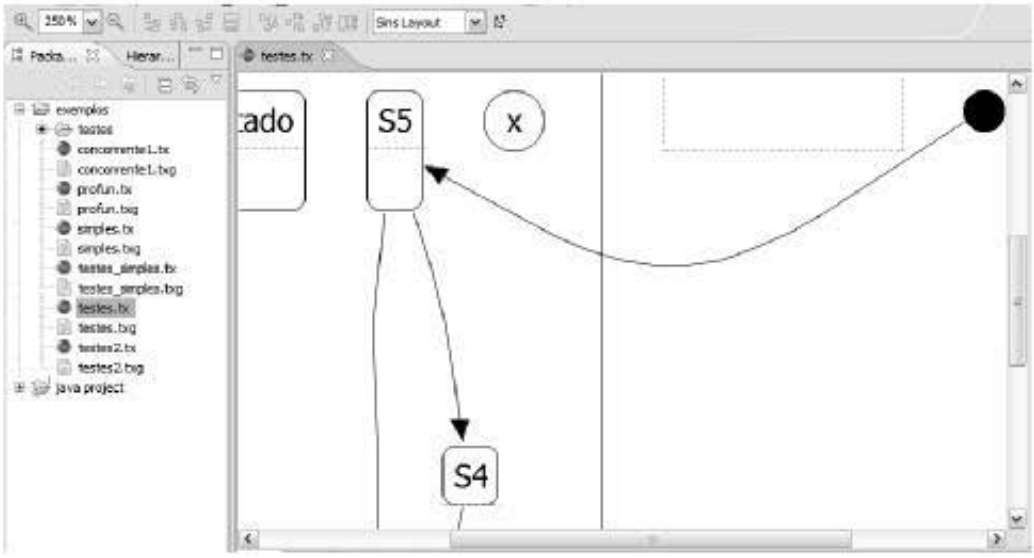


Figura 4.10: Exemplo da aplicação de zoom

4.6 Extensão à linguagem TEXchart

A leitura pelo editor de diagramas já existentes implica na ativação do recurso de layout automático, já que a linguagem TEXchart não permite a especificação da posição ou dimensões dos elementos da especificação. Da mesma forma, diagramas criados ou alterados pelo editor devem manter as posições e dimensões dos elementos.

Dadas as limitações da linguagem TEXchart, essas informações são armazenadas em um arquivo auxiliar de geometria ao arquivo TEXchart principal. Esse arquivo de geometria utiliza uma sintaxe semelhante à da linguagem TEXchart, contendo uma nova seção para armazenar essas informações.

Essa forma de armazenamento oferece as seguintes vantagens sobre uma modificação da linguagem TEXchart que exigisse a adição de uma seção no arquivo TEXchart principal:

- O padrão atual da linguagem TEXchart não precisa ser modificado.
- As ferramentas já existentes não precisam ser atualizadas nem sua documentação alterada porque não existe, no momento, ferramenta do ambiente Xchart que utilize informações sobre posicionamento dos elementos no diagrama.

- O arquivo principal, na linguagem Xchart, continua mantendo apenas o modelo do sistema reativo criado pelo projetista, separando a descrição do comportamento do sistema das informações sobre posicionamento dos elementos do diagrama Xchart, que é apenas uma das representações possíveis da especificação.

A criação e atualização do arquivo de geometria é gerenciada pelo editor, sendo transparente para o usuário projetista.

4.6.1 Formato do arquivo de geometria

O arquivo de geometria utiliza uma sintaxe semelhante à da linguagem TEXchart. Cada especificação armazenada em um arquivo TEXchart de extensão *.tx* tem os as suas informações de posição contidas num arquivo de geometria de extensão *.txg*.

Um arquivo de geometria contém apenas uma seção, iniciada com a palavra-chave *GEOMETRY*, que contém todas as informações sobre a posição e as dimensões dos elementos da especificação. A informação sobre a geometria de cada elemento é representada dentro de sua própria subseção, que inicia com o tipo do elemento (por exemplo: *XCHART*, *ACTIVITIES*, *LOCALVARS*), seguido por um ponto (.) e o identificador do elemento, como na seção *DOCUMENTATION* do arquivo TEXchart principal.

Dentro da seção equivalente a um elemento estão contidas a informação sobre a posição do elemento e, opcionalmente, sobre o seu tamanho. A posição do elemento equivale à dois inteiros correspondentes às coordenadas horizontal e vertical do centro do elemento. O tamanho do elemento, se presente, é composto de dois inteiros correspondentes à largura e à altura do elemento, respectivamente.

Um estado que tem o seu refinamento representado em outra posição da especificação tem duas subseções: uma em que é identificado normalmente, relativa à posição original do estado, onde ele é apenas referenciado, e outra em que é seu identificador é adicionado de um sufixo formado por um ponto e a palavra-chave *REFINEMENT*.

A subseção de uma transição não contém informação de tamanho, apenas de posição. Dentro da subseção são encontradas a palavra-chave *LABEL*, a posição do rótulo da transição, a palavra-chave *BENDPOINTS* e as posições dos pontos em que a transição é dobrada. A posição do rótulo é dada por um número de ponto flutuante entre 0 e 1 inclusive, onde 0 corresponde à origem da transição e 1 ao destino. A posição de uma dobra é indicada por dois pares de inteiros correspondentes às coordenadas horizontal e vertical relativas à origem e ao destino da transição, respectivamente.

Um exemplo de arquivo de geometria é mostrado na figura 4.11.

```
GEOMETRY {
  VARIABLES { 428 50 47 82 }
  EVENT.E4 { 260 174 }
  EVENT.E3 { 262 111 }
  XCHART.S1 { 41 50 367 441 }
  SUBSTATE.S1.S23 { 20 40 189 360 }
  DOTINITIAL.S1 { 332 101 }
  HISTORY.S1.H1 { 147 40 }
  TRANSITION.S1.t1 {
    LABEL 0.6750103088139964
    BENDPOINTS 18 61 -8 -50
  }
  RULES.S1.S1 { 229 40 83 83 }
}
```

Figura 4.11: Exemplo de arquivo de geometria

4.7 Conclusão

O editor proposto atende a uma série de requisitos para que seja adequado à edição de diagramas Xchart. Ele deve permitir a edição visual e direta de diagramas, fornecendo operações destinadas à modificação e visualização e utilizando como estilo de interação a manipulação direta. Se conectará ao ambiente Xchart já existente através da linguagem textual TEXchart, possibilitando a edição dos diagramas já existentes, codificados nessa linguagem. Oferecerá uma operação de layout automático de diagramas, com o objetivo de prepará-los para uso em documentos e facilitar sua compreensão e modificação. Dado o método de edição oferecido pelo editor, foi necessário estender a linguagem TEXchart para acomodar o armazenamento de posições e tamanhos dos elementos dos diagramas editados.

No próximo capítulo serão descritos a estrutura e os componentes de software utilizados para a implementação do editor Sins, assim como os algoritmos selecionados e os detalhes técnicos dessa implementação.

Capítulo 5

Detalhes da Implementação

5.1 Introdução

As partes e comportamento do editor Sins do ponto de vista do usuário foram descritos no capítulo anterior. Neste capítulo, são dados detalhes da implementação do editor: as tecnologias envolvidas na implementação (seção 5.2), os componentes do editor e suas relações (seção 5.3) e o algoritmo de layout automático de diagramas Xchart implementado no Sins (seção 5.4).

5.2 Tecnologias

Com base nos requisitos de sistema discutidos, tecnologias relativas à plataforma de desenvolvimento, linguagem, IDE e construção de parsers foram analisadas e selecionadas para o desenvolvimento do editor Sins.

Atualmente o ambiente Xchart está sendo desenvolvido sobre a plataforma Java. Essa tecnologia que tem como uma das suas principais vantagens a possibilidade de execução de uma mesma aplicação em qualquer sistema capaz de executar a Máquina Virtual Java (JVM). Além das facilidades trazidas pela independência de plataforma, a utilização do Java para o desenvolvimento do editor Sins torna disponíveis à esse processo os recursos específicos dessa plataforma, descritos em detalhes na seção 5.2.1. A JVM também está disponível em mais arquiteturas que a plataforma *.NET*, voltada aos sistemas da linha *Microsoft Windows*. Dadas essas vantagens, a plataforma Java foi a base do desenvolvimento do editor.

Existem muitos ambientes integrados de desenvolvimento disponíveis no mercado, para as mais diversas arquiteturas e plataformas, voltados a diferentes nichos de desenvolvimento, comerciais e de código livre. No caso do editor Sins, um IDE também escrito

em Java e licenciado como software livre tem grandes vantagens quanto à sua extensibilidade, principalmente com relação à este projeto. Quatro IDE com essas características se destacam: JDeveloper, BlueJ, NetBeans e Eclipse.

O JDeveloper é um IDE criado pela Oracle [67], voltado ao desenvolvimento em Java. É de uso gratuito em pequenos projetos mas seu código fonte não está disponível e não há grande suporte para extensões ao ambiente. O BlueJ é um IDE mantido em regime de software livre voltado ao ensino de Java e, principalmente, noções de orientação a objetos [68]. Seu código fonte está disponível mas não oferece facilidades para extensão e seus recursos limitados o tornam inviável para uso em desenvolvimentos de grande porte. O NetBeans é um IDE desenvolvido pela Sun [69] e recentemente teve seu código fonte licenciado como software livre. Similar ao NetBeans, o Eclipse é um IDE de código livre mantido inicialmente pela IBM. Ambos são também plataformas utilizáveis como base para outros tipo de aplicação. São facilmente extensíveis através da composição de módulos ou plugins. Entre os dois, aquele que oferece mais vantagens ao desenvolvimento do editor Sins é o Eclipse, por oferecer plugins voltados à criação de editores de diagramas e pela maior maturidade do projeto (implicando em uma comunidade de usuários e desenvolvedores maior, melhor documentação e menor número de bugs sérios). O Eclipse IDE, ambiente selecionado para ser estendido pelo editor Sins, é analisado em detalhes na seção 5.2.2.

Um dos requisitos mais importantes do editor é a integração com o ambiente Xchart através da leitura e escrita de especificações na linguagem textual TEXchart. A leitura dessas especificações exige a inclusão de um parser TEXchart no editor. Como outras ferramentas do ambiente Xchart, o editor Sins utiliza um parser gerado automaticamente a partir de uma gramática livre de contexto descrita no padrão BNF. Existe um grande número de ferramentas gratuitas ou de código livre que produzem parsers na linguagem Java, incluindo *ANTLR*, *Beaver*, *BYACC/J*, *Coco/R*, *CUP*, *JavaCC*, *jay* [70] e *SableCC*. Dentre essas, *Beaver* [71], *BYACC/J* [72] e *CUP* [73] exigem a utilização de outras ferramentas para a geração dos analisadores léxicos correspondentes, enquanto as outras produzem esses componentes automaticamente. Apenas *ANTLR* [74], *Coco/R* [75], *JavaCC* e *SableCC* integram-se ao Eclipse IDE, simplificando o desenvolvimento e a depuração do parser. Os parsers gerados pelo *SableCC* [76] constroem uma árvore sintática equivalente à sua entrada que exigiria um passo a mais para a interpretação da árvore pelo editor. Finalmente, o *ANTLR* exige a distribuição de bibliotecas auxiliares em conjunto com o parser gerado, enquanto o parser produzido pelo *JavaCC* independe de classes ou pacotes externos. Conclui-se que tanto o *Coco/R* quanto o *JavaCC* são adequados às necessidades do editor. Assim, o gerador *JavaCC*, descrito na seção 5.2.4, foi escolhido para o desenvolvimento do parser TEXchart.

5.2.1 Java

O editor Sins tem como base de usuários os pesquisadores e desenvolvedores que utilizam diagramas Xchart para a especificação de componentes de controle de diálogo de interfaces de usuário e de sistemas reativos em geral. A plataforma Java apresenta-se como uma das soluções preferidas para a implementação de ferramentas de pesquisa científica e de desenvolvimento [77, 78].

Java é um exemplo de solução independente de plataforma para o desenvolvimento de aplicações. A plataforma Java [79] oferece um conjunto de recursos comuns aos sistemas computacionais, representados pela máquina virtual Java (*JVM–Java Virtual Machine*) e pelas suas interfaces de programação padrão conhecidas como *Java Application Programming Interface*. As aplicações desenvolvidas nessa plataforma são compiladas para a arquitetura da JVM, cuja especificação é padronizada e independe do sistema onde a aplicação é executada. O código objeto da aplicação (*bytecode*) é interpretado e executado por uma implementação da JVM que executa diretamente sobre o sistema operacional da máquina. Os recursos da máquina são acessados através da Java API, que por sua vez é composta por dois conjuntos de interfaces: interfaces básicas, presentes em todas as implementações da plataforma, conhecida como *Java Core API*, e interfaces de extensão (*Standard Extension API*), cuja funcionalidade depende das implementações instaladas na máquina. A arquitetura da plataforma Java pode ser visualizada na figura 5.1.

As vantagens da plataforma Java para o desenvolvimento de aplicações, incluindo a independência de plataforma, a gerência automática de memória, o bom desempenho no processamento, a existência de bibliotecas padrão para criação de interfaces de usuário e as facilidades de orientação a objetos da linguagem Java levaram à sua escolha para o desenvolvimento recente do ambiente Xchart (como o gerente de distribuição do ambiente Xchart [62]). Assim, torna-se natural a escolha dessa plataforma para o desenvolvimento de um editor de especificações que se integra ao restante do ambiente Xchart.

5.2.2 Eclipse

Eclipse é um projeto de software livre que desenvolve uma plataforma aberta de desenvolvimento de software e um framework para a construção de aplicações. Entre os subprojetos principais estão um serviço de componentes (*Equinox*), a plataforma de desenvolvimento e ferramentas voltadas para a construção de aplicações Java e plugins que estendem a plataforma de desenvolvimento. A forma mais comum de utilização da plataforma é através do *Eclipse IDE*, um ambiente integrado para o desenvolvimento de software que inclui as principais ferramentas desenvolvidas pela comunidade.

O editor proposto foi implementado como plugin para a plataforma Eclipse porque, dessa forma, integra-se ao grande conjunto de ferramentas já existentes nessa plataforma

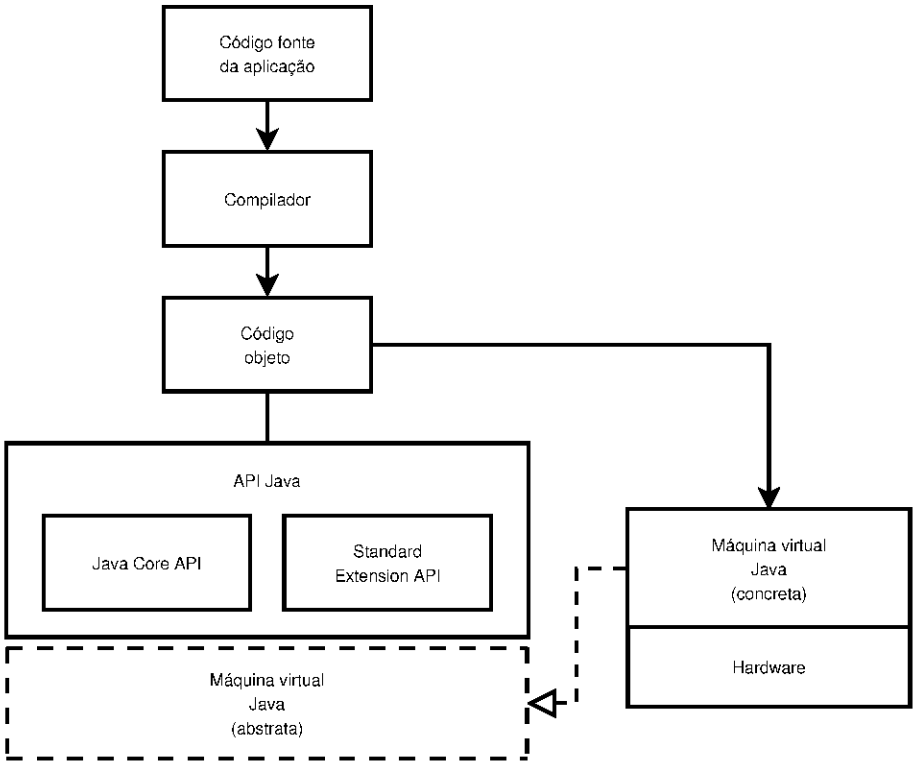


Figura 5.1: Arquitetura da plataforma Java

voltadas ao desenvolvimento de aplicações e facilita-se sua utilização em projetos de maior complexidade, incluindo aqueles onde é necessário o controle de versões, gerenciamento de equipes e documentação de *bugs* e funcionalidades implementadas. O ambiente Eclipse também inclui componentes necessários à implementação do editor, como acesso à editores de texto e código fonte padronizados e frameworks para criação de editores de diagramas.

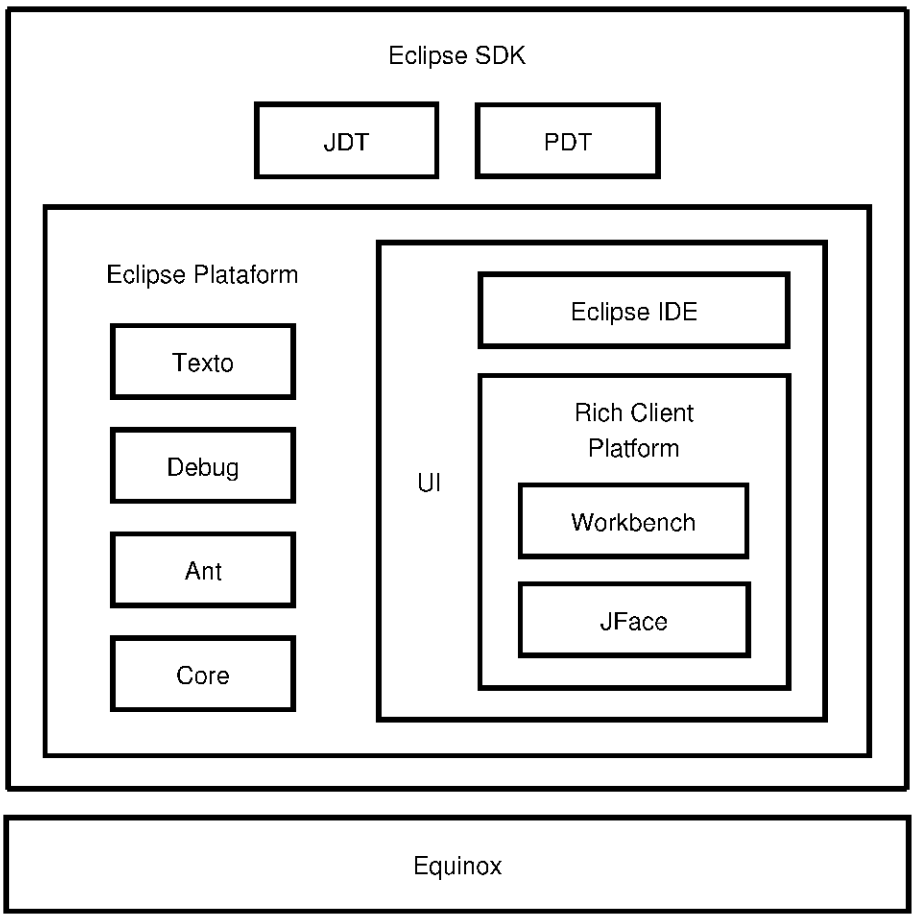


Figura 5.2: Arquitetura da plataforma Eclipse

5.2.2.1 Equinox

Equinox é um dos subprojetos do projeto Eclipse que visa implementar as especificações OSGi. Essas especificações definem um framework para a criação de aplicações extensíveis conhecidas como *bundles*, incluindo uma camada de serviços e camadas para gerenciamento de módulos, ciclo de vida das aplicações e segurança [80]. A plataforma Eclipse é construída utilizando os recursos oferecidos pelo Equinox.

5.2.2.2 Plataforma de desenvolvimento

A plataforma Eclipse compreende o conjunto de componentes que oferecem a base para a implementação de extensões na forma de plugins, incluindo componentes para gerenciamento de recursos, integração com ferramentas de controle de versões, depuração, edição e pesquisa de textos, sistema de auxílio ao usuário, instalação e atualização do software, framework de interface gráfica (*SWT Standard Widget Toolkit*) e a interface de usuário do ambiente.

O componente de interface de usuário contém uma série de componentes destinados à construção de interfaces gráficas para as aplicações criadas sobre a plataforma Eclipse. Dois desses componentes formam o framework de interface de usuário da *Rich Client Platform* (*RPC*), uma plataforma para construir aplicações cliente formada por componentes da plataforma Eclipse: *JFace* e *Workbench*.

JFace é um conjunto de componentes de interface construídos sobre o SWT que facilitam tarefas comuns na criação de interfaces de usuário, como gerenciamento de imagens e fontes, frameworks para textos, diálogos, preferências de usuário, assistentes (*wizards*) e representação de processos longos. O componente JFace também oferece a funcionalidade de ações, que encapsulam operações do usuário que podem ser ativadas de mais de uma maneira (item de menu, botão de barra de ferramentas, etc.), e visualizadores (*viewers*), baseados no padrão Modelo-Visão-Controlador, que facilitam a exibição de listas, tabelas e árvores de dados.

O componente Workbench constrói em cima do JFace para oferecer a estrutura da interface de usuário do Eclipse IDE. O objetivo do Workbench é permitir a integração adequada de diferentes ferramentas que podem ou não serem voltadas para o desenvolvimento de software. O plugin Eclipse IDE instancia e configura componentes do Workbench para apresentar um ambiente adequado para o desenvolvimento de aplicações, incluindo ferramentas voltadas para a linguagem Java e criação de plugins para a plataforma Eclipse.

5.2.2.3 Workbench

A interface de usuário do ambiente Eclipse é composta por uma ou mais janelas conhecidas como bancadas de trabalho (*workbench*). Cada janela de bancada contém uma barra de menu, uma barra de ferramentas e uma área dedicada aos documentos editados denominada página (*page*). Cada página contém um número de partes (*part*); cada parte pode ser uma visão (*view*) ou um editor (*editor*). Uma visão tem como objetivos, por exemplo, abrir editores, mostrar propriedades sobre o editor correntemente ativo ou mostrar uma hierarquia de informações. Editores são usados para editar ou navegar em um documento ou entrada. Enquanto modificações feitas em uma visão são aplicadas imediatamente, alterações feitas num editor só são salvas explicitamente, seguindo um modelo

abrir-salvar-fechar.

O Eclipse IDE pode ter sua interface de usuário estendida através de plugins instalados na plataforma. Cada janela correspondente a uma bancada de trabalho implementa a interface *IWorkbenchWindow* e contém uma barra de menu, uma barra de ferramentas, uma linha de estado (*statusline*, para mostrar o estado de elementos da bancada), uma barra de atalhos e uma página. Páginas são implementações da interface *IWorkbenchPage* e contêm partes que são especializações de classes que implementam a interface *IWorkbenchPart*. Extensões costumam criar novos tipos de partes herdando das classes *ViewPart* e *EditPart*, definindo, respectivamente, visões e editores.

O ciclo de vida de visões e editores sempre segue os mesmos passos:

- Componentes de interface de usuário SWT são criados e recursos como imagens e fontes alocados pelo método createParts.
- O recebimento do foco do usuário leva à execução do método setFocus, para que o componente de interface adequado seja ativado.
- O fechamento da visão ou do editor leva à execução do método dispose, para a liberação dos recursos alocados (imagens, fontes, etc.). Os componentes de interface de usuário (como botões, caixas de texto, etc.) são liberados automaticamente pela plataforma.

As contribuições à interface de usuário da bancada são definidas com relação a pontos de extensão. Esses pontos de extensão permitem a adição de novas funcionalidades a partes (visões e editores) existentes ou a implementação de novas partes, assim como a integração com os menus, barras de ferramentas e janela de configurações do Eclipse IDE.

5.2.2.4 Plugins

Plugins são equivalentes a bundles na arquitetura OSGi, que é a base da plataforma Eclipse. Um plugin é "...um componente estruturado que contribui com código (ou documentação, ou ambos) para o sistema e o descreve de forma estruturada" [81]. Plugins são os componentes básicos usados para construir e implementar a plataforma Eclipse.

A conexão entre plugins se dá através de pontos de extensão. Um ponto de extensão define um contrato, na forma de metadados e interfaces de programação Java, que deve ser seguido pelas suas extensões (plugins que se ligam ao ponto de extensão). Plugins também podem definir novos pontos de extensão, permitindo que novos plugins contribuam com a funcionalidade oferecida.

Os pontos de extensão existentes são registrados no registro de extensões da plataforma, que implementa a interface *IExtensionRegistry*. Dessa forma, os plugins em execução podem obter informações sobre os pontos de extensão disponíveis e quais os plugins instalados no momento.

Plugins só são executados quando sua funcionalidade é requisitada, evitando a utilização de recursos do sistema desnecessariamente. Isso é possível porque a informação sobre um plugin pode ser obtida diretamente de seu manifesto, um arquivo no formato XML (*plugin.xml*) que descreve o plugin, a quais pontos de extensão ele se liga e quais foram definidos.

Os plugins têm um ciclo de vida simples em que estes são iniciados, através de uma chamada ao método start, interagem com o resto do sistema para implementar a funcionalidade definida e, finalmente, são encerrados com uma chamada ao método stop, que deve liberar os recursos alocados e salvar o estado interno, se necessário. Esses métodos são implementados pela classe Plugin, uma classe utilitária recomendada para servir de base na definição de novos plugins.

5.2.2.5 Editores

Novos editores são ligados ao ponto de extensão *org.eclipse.ui.editors* e suas instâncias sempre estão associadas a uma entrada, definida pela interface *IEditorInput*. Apenas um editor pode estar aberto para cada entrada em uma bancada. As modificações efetuadas dentro do editor só são aplicadas à sua entrada quando o usuário executa a ação de salvar. Entre as informações definidas num plugin que implementa um editor estão os padrões de nomes de arquivos que devem ser abertos pelo editor.

Ações são a implementação de comandos disponíveis aos usuários, independentes da sua representação na interface de usuário (como ítens de menu ou botões na barra de ferramentas). Algumas ações são implementadas por vários editores e visões, como Mover, Renomear, Copiar, Colar, Desfazer, etc. Essas ações, conhecidas como ações redirecionáveis (*retargetable*), podem ser compartilhadas pelos plugins: quando o usuário executa a ação, o tratador do plugin correntemente ativo é ativado. Editores também podem contribuir com novas ações, inseridas em menus ou na barra de ferramentas e definidas em uma classe à parte que implementa a interface *IEditorActionBarContributor*.

A plataforma oferece classes e interfaces de programação para a criação de editores de múltiplas paginas, que podem exibir mais de uma visão da mesma entrada no espaço reservado ao editor. Apenas uma página, equivalente a uma visão, pode ser exibida a cada momento. Para isso, são definidas as classes *MultiPageEditorPart*, equivalente à classe *EditorPart* (que implementa a interface *IEditorPart*), e *MultiPageEditorActionBarContributor*, que equivale à classe *EditorActionBarContributor* (que implementa a interface *IEditorActionBarContributor*).

5.2.3 GEF

GEF (*Graphic Editing Framework*) é um framework para a criação de interfaces de usuário para editores gráficos. É um componente separado em dois plugins: Draw2d, para desenho e layout de elementos gráficos, e GEF, um framework interativo baseado no padrão MVC.

5.2.3.1 Draw2d

O plugin Draw2d oferece funcionalidade para o desenho vetorial de diagramas e documentos, integrada ao framework de interface de usuário SWT. Os desenhos são manipulados com base em classes que representam figuras e implementam a interface *IFigure*. Essas classes são auxiliadas por um encaminhador de eventos (*EventDispatcher*), que trata os eventos de interface gerados pelo usuário, e um gerenciador de atualização (*UpdateManager*), que controla o desenho das regiões das figuras que sofrem modificações. As figuras implementadas por essas classes são leves (*lightweight*), isto é, não correspondem a recursos do sistema gráfico do sistema operacional.

Cada figura é responsável pelo próprio desenho, que ocorre através da execução de métodos implementados pela classe correspondente, e pode conter outras figuras, que desenham a si próprias, recursivamente. Cada figura tem reservada para si e suas figuras filhas uma região onde o desenho é realizado. Qualquer desenho ou evento de interface de usuário que ocorra fora da região, realizado ou detectado pela figura ou uma das suas figuras descendentes, é ignorado. Entre os métodos envolvidos no desenho de figuras estão:

1. *paint*
   Propriedades de desenho, como fontes e cores, são definidas e o estado do mecanismo gráfico é salvo para a ser usado por todas as figuras filhas, se existirem.

2. *paintFigure*
   A figura em si é desenhada.

3. *paintClientArea*
   Desenha a área cliente, isto é, a região onde as figuras filhas serão desenhadas. Propriedades gráficas relativas apenas às figuras filhas, como mudanças no sistema de coordenadas, são definidas nesse método.

4. *paintChildren*
   Desenha cada uma das figuras filhas, sempre restaurando o estado do mecanismo gráfico antes de cada uma ser desenhada.

5. *paintBorder*
   Desenha bordas e decorações que devem aparecem por cima das figuras filhas.

A ordem em que as figuras são desenhadas, assim como a região reservada para seu desenho, influencia na detecção de eventos da interface de usuário: figuras podem ser sobrepostas a outras desenhadas anteriormente e ter prioridade no tratamento de eventos (em relação às figuras que estão escondidas sob elas).

Figuras compostas posicionam suas figuras filhas através de um gerente de layout oferecido por classes que implementam a interface *LayoutManager*. Essa classe é associada à figura composta e é consultada para reposicionar cada figura filha, levando em conta as políticas implementadas e as informações fornecidas pelas figuras filhas, como tamanhos mínimo, máximo e preferido. O cálculo do layout ocorre apenas quando as figuras são marcadas como inválidas: isso leva à invalidação de todas as figuras que contêm ou são contidas pela figura invalidada. Posteriormente, o gerente de layout percorre todas as figuras inválidas e coordena o cálculo de suas novas posições.

Mais de um sistema de coordenadas pode ser utilizado para o desenho de figuras filhas. Por padrão, todas as coordenadas são absolutas em relação ao componente de interface que contém todos as figuras, mas o sistema de coordenadas usado no layout das figuras filhas pode ser trocado pela figura que as contém. Além do sistema de coordenadas absoluto (ou herdado), o mais comum é o sistema relativo (ou local): coordenadas relativas de figuras filhas se referem à figura onde estão contidas. Coordenadas absolutas podem ser convertidas em relativas e vice-versa através de métodos declarados na interface IFigure.

O pacote Draw2d também contém funcionalidade para o desenho de conexões. Conexões são figuras que representam visualmente a ligação entre duas outras figuras. As extremidades, chamadas de origem e destino, são definidas por âncoras (classes descendentes da classe *ConnectionAnchor*), que por sua vez são associadas a figuras. Dessa forma, os pontos relativos às extremidades podem ser calculados a partir dos outros pontos da conexão e das figuras envolvidas, considerando, por exemplo, a intersecção entre a conexão e a borda das figuras. Conexões também podem conter pontos internos de dobra (*bending points*), permitindo o posicionamento livre da conexão, sempre de acordo com a implementação de um roteador implementado numa classe descendente da classe *ConnectionRouter*. Como qualquer figura, uma conexão pode conter figuras filhas, correspondentes, por exemplo, a rótulos ou setas e símbolos nas extremidades.

5.2.3.2 GEF

O plugin GEF oferece recursos para a construção de visualizadores e editores de quaisquer modelos ou informações, gerenciando sua exibição e a interação do usuário com os modelos. O GEF utiliza a funcionalidade oferecida pelo Draw2d.

O framework GEF baseia-se no padrão MVC: o modelo pode ser qualquer informação a ser exibida ou editada (representada através de classes fornecidas pelo desenvolvedor), a visão corresponde à representação visual desse modelo (composta de figuras Draw2d)

e o controlador gerencia a modificação do modelo e a atualização da visão de acordo com essas modificações (formado por classes fornecidas pelo desenvolvedor chamadas de Editparts). Cada um desses três tipos de elementos forma uma hierarquia de objetos; as três hierarquias se relacionam intimamente, representando os três aspectos do sistema criado.

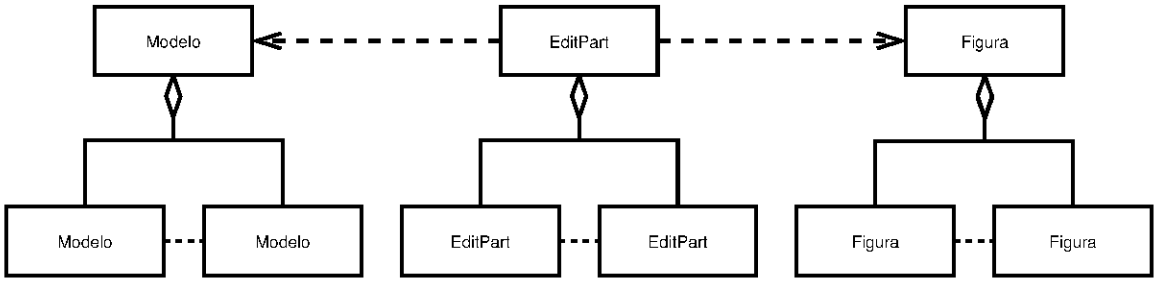


Figura 5.3: Hierarquia de objetos em um editor GEF

As visões são representadas em componentes de interface de usuário fornecidos pela framework GEF e conhecidos como visualizadores (*viewers*). A uma instância de visualizador são associados o objeto raiz do modelo e um objeto que segue o padrão factory, responsável por criar as editparts referentes a cada objeto do modelo. Quando o visualizador é ativado, as editparts são criadas a partir do modelo e cada editpart constrói sua visão através de figuras Draw2d que são apresentadas pelo visualizador. A editpart relativa ao objeto raiz do modelo deve criar uma figura especial como sua visão, conhecida como camada (*layer*), sobre a qual todas as outras figuras serão desenhadas. A camada a ser criada depende do espaço disponível para os desenhos: a classe *FreeformLayer* permite figuras com coordenadas negativas, diferentemente da classe *Layer*. No primeiro caso, a editpart raiz deve herdar de *ScalableFreeformRootEditPart*; caso contrário, de *ScalableRootEditPart*.

Editparts são fornecidas pelo desenvolvedor, herdando da classe *AbstractGraphicalEditPart* e fazendo o papel do controle do padrão usado pelo GEF. As responsabilidades de editparts incluem criar e manter sua visão, suas editparts filhas, editparts de conexão e apoiar a edição do modelo.

A visão é criada com a execução do método *createFigure* e a atualização da visão para refletir mudanças nas propriedades do modelo associado à editpart é implementada no método *refreshVisuals*; ambos os métodos devem ser implementados pelo desenvolvedor.

Durante a criação da editpart, o método *getModelChildren* é chamado e deve retornar uma lista com os objetos do modelo que darão origem às suas editparts filhas.

Editparts referentes à conexões são compartilhadas entre as duas extremidades da conexão e são criadas pelos métodos *getModelSourceConnections* e *getModelSourceConnections* (extremidade de origem e de destino, respectivamente) das editparts das extremi-

dados; o GEF garante a unicidade das editparts conexões criadas dessa forma. A editpart relativa à conexão deve necessariamente criar sua visão baseada em uma figura Draw2d descendente da classe *Connection*.

A modificação do modelo é intermediada pelas editparts, que respondem aos eventos de interface de usuário representados por requisições (objetos da classe *Request*) com comandos (objetos da classe *Command*) que realizam as alterações quando executados. Esse processo de resposta é delegado a instâncias de classes descendentes de *EditPolicy*, fornecidas pelo desenvolvedor ou disponíveis no GEF e criadas pela editpart no método *createEditPolicies*. As políticas de edição são instaladas em papéis pré-definidos pelo framework, relativos ao tipo de elemento da visão envolvido.

As requisições podem indicar vários tipos de interação com editparts, que podem atuar em dois papéis: fonte (*source*) e alvo (*target*). Editparts fonte correspondem à seleção de editparts corrente no momento da interação; a editpart alvo é aquela apontada pelo mouse no momento, se existir. Pelo menos uma editpart está selecionada a cada momento, seja a raiz ou qualquer combinação de editparts não-raízes.

O framework também oferece a funcionalidade de ferramentas para facilitar a interação do usuário com o modelo visualizado. Essas ferramentas são similares a máquinas de estado, executando uma ação dependendo do estado do sistema e da sua ativação. Entre as responsabilidades de ferramentas estão a obtenção e execução de comandos a partir de editparts, atualização do cursor do mouse e exibição de feedback por parte das editparts. Classes relativas à ferramentas são instanciadas pelo desenvolvedor, formando uma árvore que é visualizada através de um dos visualizadores de paleta fornecidos pelo GEF (que utilizam a classe *PaletteViewer*).

5.2.4 JavaCC

JavaCC é uma ferramenta para a geração de analisadores sintáticos $LL(k)$ a partir de uma especificação de gramática [82]. Para a geração do parser é necessário um arquivo fonte EBNF contendo as definições das gramáticas envolvidas. Esse arquivo é processado pela ferramenta, que gera o código fonte do reconhecedor na linguagem Java.

As definições de gramáticas aceitas pelo JavaCC têm formato similar à de ferramentas de mesmo propósito como o *yacc* e o *bison*. Cada arquivo de definições de gramáticas, cujo nome deve terminar com a extensão *.jj*, contem a especificação de uma gramática. O arquivo inicia com uma seção opcional contendo as opções relativas à geração do parser. Cada gramática dá origem a um parser cuja classe é declarada entre uma linha *PARSER_BEGIN(Classe)*, onde *Classe* é o nome da classe do parser, e *PARSER_END(Classe)*. Em seguida, as produções da gramática são listadas.

As produções podem definir um trecho de código Java responsável por reconhecer

alguma cadeia (indicada pela palavra-chave *JAVACODE*), uma expansão no formato EBNF, uma expressão regular a ser reconhecida pelo analisador léxico gerado ou declarações para o gerenciador de tokens. Tanto produções do tipo JAVACODE quanto expansões EBNF reproduzem a estrutura recursiva do parser gerado: iniciam com a declaração do método Java correspondente à produção. Após essa declaração, um trecho arbitrário de código Java pode ser especificado para que seja executado sempre que a regra for usada na análise. No caso de produções EBNF, o nome da produção equivale ao identificador do método Java declarado no seu início e a produção é terminada com a expansão da regra.

Regras podem especificar mais de uma alternativa para a estrutura de átomos a ser reconhecida; para cada alternativa, um trecho de código a ser executado no momento do reconhecimento da alternativa pode ser definido. Referências a caracteres literais podem ocorrer apenas em regras de análise léxica; seqüências de caracteres nessas regras são interpretadas como seqüências de caracteres literais. Seqüências em regras de parser implicam na criação de um token equivalente.

Uma regra de expressão regular é composta pelo seu tipo (*TOKEN*, *SPECIAL_TOKEN*, *SKIP* ou *MORE*) e das expressões regulares a serem reconhecidas. Símbolos do tipo *TOKEN* são reconhecidos na entrada e passados diretamente ao parser. Os do tipo *SPECIAL_TOKEN* são não são reconhecidos na entrada mas podem ser ligados a outros símbolos na sua declaração, sendo transmitidos em conjunto com esses símbolos durante a análise sintática. Símbolos *SKIP* são ignorados pelo parser e símbolos *MORE* são acumulados em um buffer até que um símbolo de outro tipo seja recebido, quando são ignorados ou combinados e interpretados pelo parser. Após a especificação das alternativas a serem reconhecidas, um trecho de código Java pode ser especificado para que seja executado quando o símbolo é reconhecido.

Vários operadores podem ser usados na expansão de expressões regulares e regras EBNF da gramática. Entre os operadores disponíveis estão as subregras (especificadas entre parênteses), alternativas (/), opcional (*?* ou entre colchetes), fecho (*), fecho positivo (+), negação do reconhecimento de caracteres de uma lista em uma expressão regular (~) e intervalo entre caracteres em uma expressão regular (-). A expansão pode conter um número de elementos, incluindo a indicação do *lookahead* a ser usado pelo parser para reconhecer a produção, um bloco de código Java a ser executado quando o parser reconhece a produção até o ponto em que o código foi inserido, uma referência a um símbolo terminal (expressão regular) ou a um não-terminal (regras EBNF). Referências a símbolos podem ser precedidas por um identificador Java e um símbolo de igualdade (=), indicando que o resultado do método correspondente ao símbolo será atribuído à variável especificada. Como regras EBNF correspondem a métodos na classe do parser, referências a essas regras podem receber argumentos, tal como uma chamada de função.

A partir de uma gramática em um arquivo de extensão *.jj*, o JavaCC gera o código fonte de uma classe correspondente ao parser com o nome indicado na especificação da gramática. Também são gerados uma interface contendo constantes usadas pelo parser, uma classe responsável pela análise léxica da entrada e classes padronizadas relativas à definição de símbolo (classe *Token*), fluxo de caracteres (classe *SimpleCharStream*) e erros léxicos (classe *TokenMgrError*) e sintáticos (classe *ParserException*). Cada regra implica na geração de um método na classe correspondente, como definido no código fonte da gramática.

5.3 Componentes

O editor Sins oferece a funcionalidade de edição visual de diagramas Xchart e textual do código fonte TEXchart correspondente. Foi implementado como plugin para o Eclipse IDE, integrando-se naturalmente à interface de usuário oferecida pelo ambiente. A funcionalidade oferecida pelo editor foi dividida em um número de componentes especializados, representados na figura 5.4 e descritos nas próximas seções.

Existem duas formas de iniciar o editor: através de um assistente (*wizard*) ou através da abertura de um arquivo TEXchart (extensão *.tx*). O assistente é registrado junto ao ponto de extensão correspondente no Eclipse de forma a ser listado quando a função de iniciar assistente do Eclipse é ativada. Quanto ativado, o assistente auxilia o usuário na criação de um novo arquivo TEXchart e cria uma instância do editor de especificação. Alternativamente, o usuário pode ativar a abertura de um arquivo TEXchart já existente entre os arquivos listados através da interface de usuário do Eclipse; esse processo também cria uma instância do editor de especificação.

O resto dos componentes se organiza em um padrão Model-Visão-Controlador onde o editor de especificação é o controlador, a especificação é o modelo e os outros editores são as visões.

5.3.1 Editor de especificação

O editor de especificação faz o papel do controle do padrão MVC. Suas responsabilidades incluem criar instâncias do modelo e das visões e gerenciar as ligações com o resto do ambiente Eclipse e entre as visões e o modelo.

Na sua inicialização, o editor de especificação cria uma instância de cada editor correspondente à cada visão disponível do modelo. Após isso, o modelo é instanciado a partir do conteúdo do arquivo de entrada através do gerente de conteúdo.

Cada visão é atualizada quando é ativada, com uma chamada ao método *updateViewer*. A existência de modificações desde a última vez que a especificação foi salva são sinalizadas

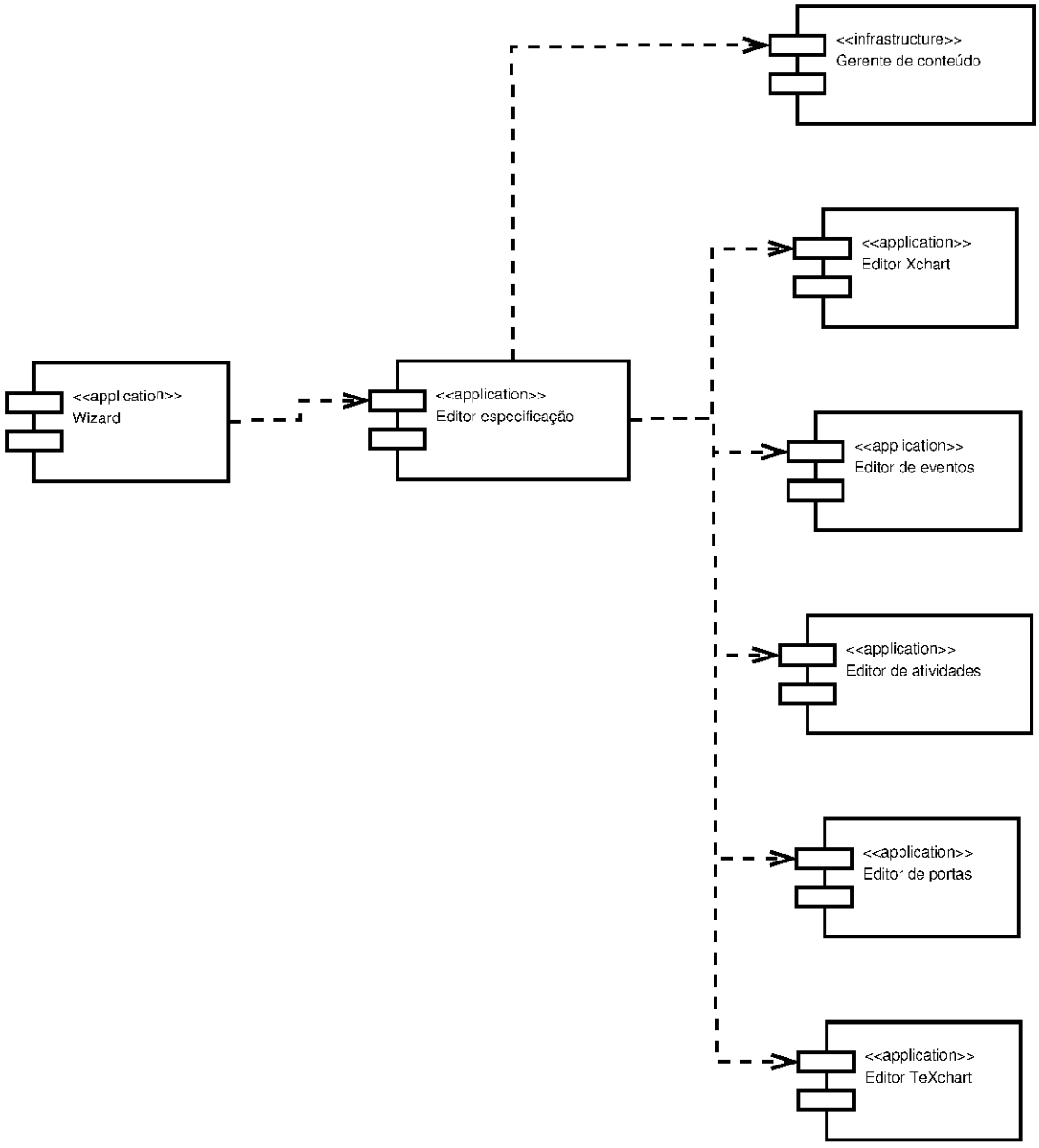


Figura 5.4: Componentes principais do editor Sins

ao editor de especificação para que o Eclipse indique em sua interface de usuário essa condição e proteja as modificações com confirmações de fechamento do editor. O estado do editor quanto à esse controle é retornado pelo método *isDirty*. Quando a especificação é salva, todos os editores são instruídos a limpar esse sinal através do método *resetDirty*.

Como existe a necessidade de detectar alterações no modelo quando da ativação de cada visão, mesmo que essas alterações tenham sido salvas em arquivo, cada visão controla as modificações feitas desde a sua última ativação. Esse controle é retornado pelo método *hasChanged* e reiniciado no momento da ativação da visão com o método *resetChanged*.

Com a ativação de uma visão diferente, a visão corrente é desativada. O modelo é atualizado pelo método *updateModel* da visão desativada. Tanto na atualização da visão quanto na do modelo, este é acessado através de métodos do gerente de conteúdo, ao qual é delegada a tarefa de manter sincronizados o modelo e sua representação textual na linguagem TEXchart.

Ações padronizadas de cada editor, como operações de deletar e desfazer alteração, são obtidas pelo editor de especificação para serem integradas à interface de usuário do Eclipse através do método *getAction* de cada editor. Essas ações são ligadas ao ambiente por meio da classe auxiliar *SpecificationEditorContributor*. Contribuições à barra de ferramentas do editor Sins são inseridas a cada ativação de uma visão através do seu método *contributeToToolbar*.

Novos editores em geral que modifiquem o modelo devem estender a classe *ModelEditor*, implementando os métodos *createViewer* (que cria um viewer para o editor) e *getSelectionProvider* (que retorna um provedor de informações sobre seleção de elementos no editor).

5.3.2 Editores gráficos

Ambos os componentes editor Xchart e editor de eventos oferecem visões para interação com o modelo da especificação. O comportamento desses componentes é bastante similar, podendo ser descrito como um componente de editor gráfico.

A responsabilidade desse componente é oferecer uma representação visual da especificação e permitir a modificação da especificação através da manipulação desses elementos gráficos. Esse componente também se organiza segundo o padrão MVC: o modelo da especificação faz o papel do modelo também nesse componente, o controle é papel de uma hierarquia de classes que herdam da classe *EditPart* e a visão corresponde às figuras que herdam da classe *Figure*.

As partes de edição são criadas a partir de objetos do modelo por um objeto que implementa o padrão fábrica, como descrito na seção 2.6.3.2 e representado na figura 5.5. As figuras correspondentes são criadas durante a inicialização de cada parte e são

exibidas em um visualizador que é instância da classe *ScrollingGraphicalViewer*, tornando-se disponíveis para manipulação pelo usuário. Essas interações são tratadas por políticas de edição gerenciadas pelas partes de edição, gerando instâncias de comandos que, quando executados, modificam o modelo. Modificações no modelo são sinalizadas às partes de edição correspondentes, para que estas atualizem as figuras que as representam.

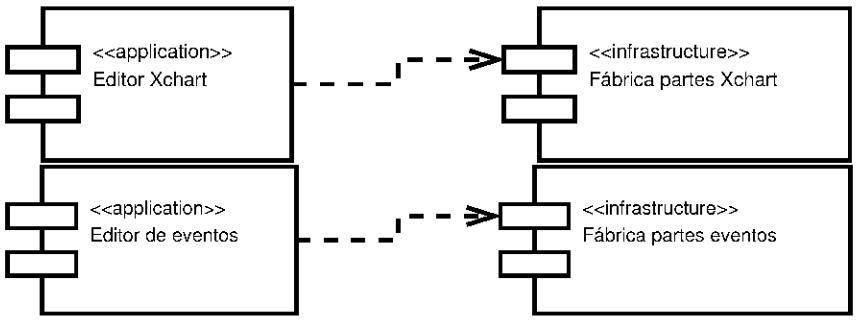


Figura 5.5: Componentes dos editores gráficos

Novos editores gráficos que utilizem o framework GEF devem estender a classe *GraphicalEditor*, implementando os métodos *createPaletteRoot* (para criar os componentes da barra de ferramentas do editor), *createPartsFactory* (para criar a fábrica de partes GEF) e os métodos de acesso à propriedade *hasGeometry* (que consultam o gerente de conteúdo para definir se uma aplicação do layout automático é necessária).

5.3.3 Editor de listas

Tanto o editor de atividades quanto o editor de portas têm em comum a modificação de uma lista de identificadores. Sendo assim, ambos os componentes podem ser descritos como editores de lista.

Esse componente também utiliza o padrão MVC. O modelo corresponde à especificação, o controlador é o componente e a visão é dada por um objeto da classe *ListViewer* do framework de interface de usuário do Eclipse conhecido como *JFace*. Essa visão exige um adaptador que forneça o texto e as imagens correspondentes aos elementos da lista, isolando sua representação do seu conteúdo. Nos editores de atividades e de portas, esse adaptador é associado à lista de strings correspondente do modelo. Cada modificação leva à criação de um comando que é executado através da pilha de comandos do editor ligada ao modelo.

Editores de listas de nomes devem estender a classe *ListEditor*, implementando os métodos que criam os controladores de adição, edição, atualização e verificação de nomes, assim como os métodos *getList* (que retorna a lista de nomes dentro do modelo a ser

editada) e *getNameServiceId* (que retorna o nome do serviço de nomes dentro do gerente de nomes).

5.3.4 Editor TEXchart

O componente editor TEXchart apresenta como visão do modelo um elemento textual padrão do ambiente Eclipse, herdando da classe *TextEditor*. O modelo também é a especificação, mas esta é acessada indiretamente porque essa visão exibe e manipula apenas texto.

Como herda da classe padrão de editores de texto do Eclipse, esse componente apresenta o mesmo comportamento dos outros editores. Ações como desfazer e refazer alterações, imprimir e marcar pontos do texto estão incluídas na classe original.

Editores de texto padrão também exibem o conteúdo do arquivo de entrada após a inicialização. Dessa forma, quando a especificação é salva e o método *resetDirty* do editor TEXchart é chamado, este é apenas forçado a atualizar seu conteúdo a partir do arquivo de entrada.

5.3.5 Gerente de conteúdo

O gerente de conteúdo (*ContentManager*) é o componente responsável por criar instâncias do modelo e gerenciar a tradução entre a especificação e o código fonte TEXchart. Seus subcomponentes são representados na figura 5.6.

Esse componente utiliza os serviços dos parsers e geradores TEXchart e de geometria, por intermédio da classe auxiliar *ContentFactory*, para manter a sincronia entre o modelo e o código fonte correspondente. Seu comportamento baseia-se na constatação de que existem duas grandes categorias de editores no Sins: editores de modelo (descendentes da classe *ModelEditor*) e o editor TEXchart (descendente da classe padrão *TextEditor*). Editores de modelo são capazes de manipular a instância corrente da especificação diretamente, enquanto que o editor TEXchart manipula o código fonte correspondente. Esse código fonte pode ser obtido do gerente de conteúdo, no caso de uma ativação do editor TEXchart, ou do arquivo de entrada associado ao editor, durante sua inicialização ou após a operação de salvar.

Inicialmente, o gerente de conteúdo é inicializado com o arquivo de entrada aberto no editor principal. Essa configuração, registrada com o método *setInput* tornam o modelo inválido porque fora de sincronia. Quando um editor de modelo é ativado, uma chamada ao método *getModel* leva à interpretação do conteúdo do arquivo de entrada pelos parsers TEXchart e de geometria e à criação de uma nova instância do modelo. Durante a desativação de um editor de modelo, na qual é executado o seu método updateModel, modificações são sinalizadas ao gerente de conteúdo através do método *setSourceCodeSynchronized*.

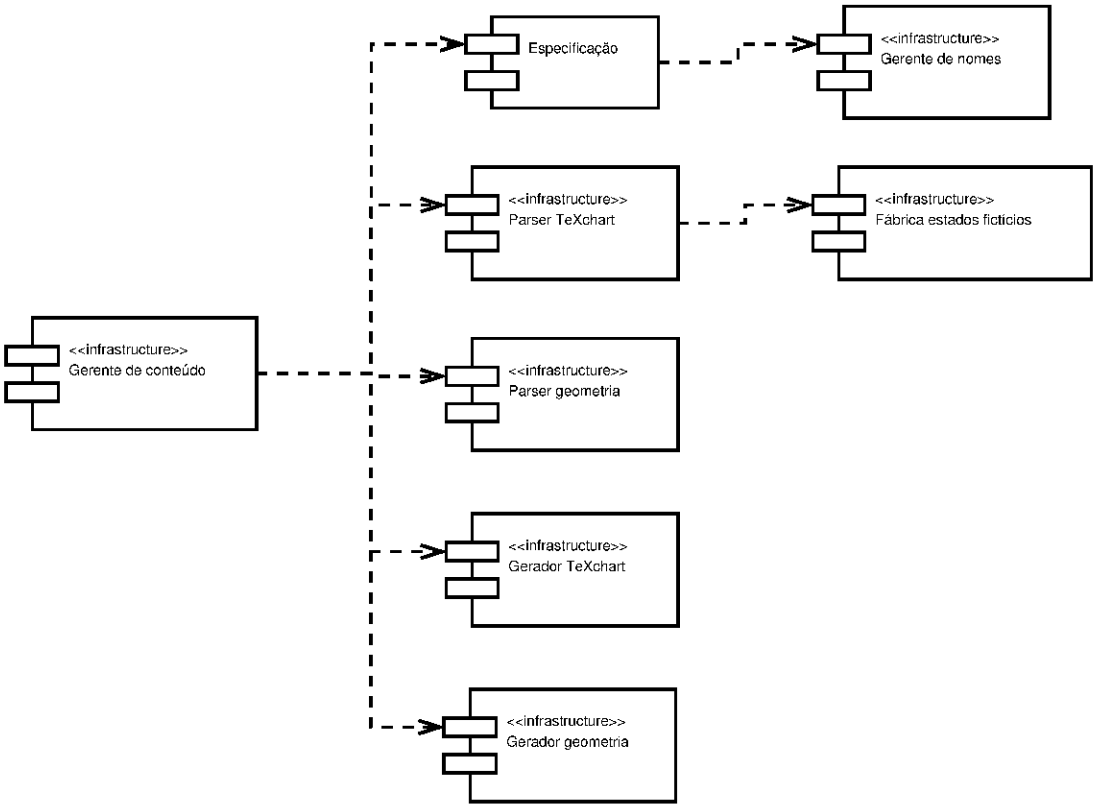


Figura 5.6: Componentes do gerente de conteúdo

Quando o editor TEXchart é ativado, seu conteúdo pode ser atualizado (devido às eventuais alterações na especificação) com uma nova versão do código fonte fornecida pelo método *getTexchart* a partir do gerador TEXchart. Mudanças efetuadas no código fonte dentro do editor TEXchart são ao gerente de conteúdo com o método *setTexchart*. Esse método utiliza o parser TEXchart para criar uma versão atualizada do modelo, sincronizando-o ao novo código fonte.

O gerente de conteúdo também é responsável pela operação de salvar a especificação em arquivo. O código fonte da especificação é obtido do método *getTexchart* e a geometria do método *getGeometry*. As chamadas ao método getTexchart sempre sinalizam o código fonte como sincronizado, já que o editor TEXchart tem seu conteúdo atualizado após a gravação da especificação através do seu método *resetDirty* e, nesse caso, não precisa obter a nova versão do texto quando é ativado.

5.3.5.1 Parsers

Ambos os parsers (TEXchart e geometria) são gerados pela ferramenta JavaCC. Os parsers gerados são do tipo recursivo, possibilitando a passagem de parâmetros às regras e o retorno de valores destas. Regras criam e retornam instâncias correspondentes aos elementos reconhecidos na entrada sempre que possível. Nos casos em que filhos só podem ser adicionados ao objeto que os contém através de seus métodos, as regras responsáveis pela criação dos filhos recebem como parâmetro uma referência ao objeto pai para que a construção dos objetos se dê corretamente.

O subcomponente fábrica de estados fictícios (*DotStateFactory*) é utilizado pelo parser TEXchart para gerenciar a criação de estados fictícios dentro de estados do modelo. Adotou-se a convenção de que cada estado pode conter apenas um estado inicial e um estado final. Essa convenção é mantida através da fábrica de estados fictícios: o método *createDotState* retorna o estado fictício contido no estado especificado, criando-o se necessário. Duas fábricas, uma para estados iniciais e outra para estados finais, são mantidas pelo parser.

Devido às facilidades oferecidas por parsers recursivos, métodos do parser que implementam o reconhecimento de regras são aproveitados para a verificação da sintaxe de regras e declarações de variáveis durante a edição na visão Xchart.

5.3.5.2 Geradores

Ambos os geradores são responsáveis pela produção do código fonte correspondente ao modelo. O gerador TEXchart gera o código fonte principal, equivalente à especificação e os diagramas que ela contém. O gerador de geometria produz o código fonte que descreve as posições e dimensões dos elementos gráficos da especificação.

Assim como acontece com o parser TEXchart, métodos do gerador TEXchart são usados pela visão Xchart para a exibição de regras e declarações de variáveis a partir do modelo.

Novos geradores podem ser construídos a partir da classe *Generator* ao implementar o método *generate* (que escreve o texto a partir de uma instância do modelo); métodos dessa classe como *write* e *writeln* podem ser utilizados nessa tarefa.

5.3.6 Especificação

O modelo mantém os dados da especificação Xchart correntemente editada no editor. Cada elemento de uma especificação é representado por uma classe, equiparando-se à estrutura de diagramas na linguagem Xchart. Uma instância da classe *Specification*, equivalente à uma especificação Xchart, contém os estados e eventos raiz da especificação e as listas de atividades e de portas. Todos os elementos Xchart são representados por classes que herdam da classe *Element*, que introduz um campo relativo à posição do elemento no diagrama. Esses elementos são eventos (classe *Event*) e duas hierarquias correspondentes aos outros elementos hierárquicos (*HierarchicalElement*) e às conexões (*AbstractModelConnection*), que dão origem às transições e às relações de subtipo envolvendo eventos.

Elementos hierárquicos se dividem em três classes: elementos passíveis de serem conectados a outros elementos (*AbstractConnectable*), conjuntos de declarações de regras ou variáveis (*Declaration*) e ítens de declarações (*DeclarationItem*). Elementos conectáveis contêm referências às suas conexões, que implementam a interface *ModelConnection*, e dão origem aos estados e aos eventos. Símbolos de história são considerados estados terminais (impossibilitados de conter outros estados), assim como os estados fictícios.

Para gerenciar a formação de nomes durante a criação de novos elementos e impedir a criação de elementos com nomes repetidos, as classes do modelo utilizam os serviços do gerente de nomes. Um gerente de nomes é atribuído a cada classe de elementos cujos nomes precisam ser controlados: atividades, portas, eventos, símbolos de história, estados fictícios e reais.

5.3.7 Layout automático

Os algoritmos de layout automático são aplicados a partir dos editores gráficos e implementam a interface *GraphLayout*, que contém apenas um método: *visit* (que recebe uma instância da classe *CompoundDirectedGraph* e modifica os valores das posições e tamanhos dos elementos do grafo).

O algoritmo de layout do editor Sins é implementado no pacote Java sins.layout.sins, que contém a classe *SinsLayout*. As camadas utilizadas pelo algoritmo são instâncias de classes que descendem de *AbstractLayerList* e contêm instâncias de *NodeItem*. Ítens são associados a um vértice de um grafo (classes *Node* ou *Subgraph*) e a um valor de

ponto flutuante (utilizado para ordenar camadas e vértices). Também contêm métodos utilitários para obter vizinhos, ancestrais, descendentes e baricentros. A posição dos vértices é armazenada numa instância da classe *Point*.

A classe *UnitaryLayerList* contém instâncias de *UnitaryNodeItem* e sua especialização, *SubgraphItem*; tais objetos admitem exatamente um elemento por camada. Essa classe é responsável pela fase de hierarquização e ordenação de camadas do algoritmo. As camadas são comparadas, durante a ordenação, pela classe *UnitaryLayerComparator*.

As fases seguintes do algoritmo são implementadas pela classe *LayerList*, que contém instâncias de *LayerItem*. Aqui, as camadas podem ter mais de um elemento, todos implementando a interface *NodeItem*. Ciclos são encontrados através da classe *DeepFirstSearch*. A redução de cruzamentos é implementada pela classe CrossingSorter, comparando vértices através da classe *NodeComparator*. Vértices de subgrafos são instâncias da classe *BorderNodeItem*, vértices intermediários da classe *IntermediateNodeItem* e os outros da classe *NormalNodeItem*. Durante o posicionamento de vértices, estes são ordenados segundo sua prioridade com o auxílio da classe comparadora *PriorityComparator*; o posicionamento é feito pela classe *PositionUpdater*.

5.4 Layout automático

5.4.1 Introdução

O editor Sins oferece a funcionalidade de layout automático dos diagramas editados, posicionando e dimensionando seus elementos com o objetivo de aumentar a legibilidade das especificações, assim como atender a critérios estéticos. Essa função é aplicada a pedido do usuário ou quando um arquivo contendo uma especificação é aberto sem que o arquivo de geometria seja lido com sucesso.

Diagramas em geral e especificações codificadas na linguagem Xchart em particular são grafos e, como tal, podem se beneficiar de algoritmos para o layout de grafos. Um algoritmo de layout automático de grafo tem como entrada um grafo e gera como saída um mapa correspondente, isto é, uma representação visual da entrada.

Os algoritmos de layout existentes podem ser classificados quanto à entrada esperada, quanto às métricas priorizadas ou às características do mapa gerado e quanto à técnica usada pelo algoritmo para criar o mapa, como descrito na seção 2.5.2.

5.4.2 Análise do problema

Diagramas Xchart são dígrafos compostos. A relação de inclusão é representada com o desenho dos vértices dentro dos limites visuais dos vértices ancestrais. É comum que as

arestas conectem vértices filhos de pais distintos ou vértices de mesmo ancestral mas diferentes profundidades. Ciclos são freqüentes, inclusive arestas que conectam um vértice a si mesmo. A maioria das arestas contém rótulos com textos que podem ser significativamente extensos. Arestas podem conectar um vértice a seus ancestrais ou seus descendentes.

Nos mapas gerados pelos usuários, existe a tendência à economia de espaço em detrimento da consistência no posicionamento dos elementos. Essa economia de espaço é buscada com a compactação de estados em uma matriz ortogonal, muitas vezes modificando o fluxo das arestas no meio de uma seqüência de vértices. A representação de arestas com dobras através de curvas é a regra nesses mapas, com poucos exemplos de utilização de segmentos de retas ou ângulos retos. Os ciclos são posicionados em layouts circulares, com as arestas desenhadas formando uma elipse de forma a refletir essa circularidade.

A grande desvantagem da busca excessiva por economia espaço é a diminuição da legibilidade do diagrama, principalmente naqueles com um número significativo de elementos. As relações entre os elementos do grafo perdem-se com a utilização de um layout ortogonal, exigindo que o usuário consulte freqüentemente as arestas para montar um modelo mental da estrutura do diagrama. Outra dificuldade do layout ortogonal é a necessidade de maior número de dobras nas arestas, o que também diminui a legibilidade.

O algoritmo do editor Smart, assim como o algoritmo original de Sugiyama, apresenta alguns problemas com relação aos mapas gerados. Como o algoritmo destaca a hierarquia inerente ao dígrafo e desenha arestas apenas entre níveis diferentes, são necessários vários níveis que ocupam um espaço significativo e tornam a relação entre a altura e a largura do mapa bastante desigual.

Outra desvantagem é o tratamento dado aos ciclos. Esse algoritmo aceita grafos com ciclos como entrada mas esses ciclos são desfeitos no início do algoritmo, que essencialmente ignora sua existência. Assim, ciclos no diagrama são indistinguíveis do fluxo acíclico, diferentemente dos mapas criados manualmente pelos usuários.

Também nesse algoritmo, a fase de normalização leva à criação de estados fictícios fora dos estados para evitar o cruzamento entre arestas e os limites superiores e inferiores dos estados ancestrais das suas extremidades. Isso leva à criação de dobras desnecessárias nas arestas, diminuindo a legibilidade e aumentando a área ocupada pelo mapa. Essas dobras externas aos estados também causam, durante a etapa de layout métrico, o desalinhamento entre os vértices de origem e de destino das arestas dobradas, diminuindo significativamente a simetria do mapa gerado.

5.4.3 Algoritmo proposto

O algoritmo proposto, implementado no editor Sins, tem como prioridade a legibilidade do mapa gerado e, em segundo lugar, a criação de mapas similares aos criados manualmente

pelos usuários. Para o primeiro objetivo, foram definidos princípios a serem seguidos pelo algoritmo, como a consistência na apresentação das estruturas do diagrama. Com relação ao segundo objetivo, ao qual foi dada menor prioridade, foram analisados exemplos de diagramas Xchart retirados da especificação da linguagem Xchart [13], assim como de exemplos da referência da linguagem [83].

Considerando as propriedades inerentes aos diagramas Xchart descritas na seção anterior, são definidos princípios de usabilidade como os descritos por Nielsen [84] para a geração de mapas legíveis a partir desses diagramas. Dentre esses princípios, aqueles que são passíveis de aplicação incluem a consistência e o uso de padrões, reconhecimento em lugar de recordação, estética e design minimalista e visibilidade do estado do sistema (no sentindo de minimizar a distância entre a representação do diagrama e o modelo mental do usuário).

Levando em conta esses princípios, o algoritmo procura representar, de forma consistente e claramente diferenciada, as estruturas do grafo. Uma dos tipos reconhecidos de estrutura é o componente conexo cujas transições formam um fluxo unidirecional (acíclico). Outro tipo de estrutura é o ciclo formado por estados do diagrama. Por fim, estados fictícios são também estruturas diferenciadas porque representam o início ou o fim de um fluxo de transições. Nota-se que o reconhecimento de estruturas no grafo não implica em seu particionamento: um mesmo elemento pode pertencer a mais de uma estrutura.

Foram definidas heurísticas para cada uma das estruturas reconhecidas. Fluxos unidirecionais são representados em camadas, hierarquicamente, orientados horizontalmente ou verticalmente e, respectivamente, da esquerda para a direita ou de cima para baixo. Na orientação horizontal, todas as arestas apontam para a direita; na vertical, todas as arestas apontam para baixo.

Ciclos são representados em um layout aproximadamente circular, posicionando os vértices participantes ao longo de uma circunferência imaginária. Este é o único caso onde as arestas podem apontar em sentido contrário ao natural do mapa: de baixo para cima ou da direta para a esquerda.

Estados fictícios são sempre posicionados no início ou no fim do fluxo do qual fazem parte. No caso de estados iniciais, sempre no local mais ao topo ou à esquerda possível no mapa; estados finais são posicionados sempre no local mais baixo ou à direita possível.

Além disso, a partir das informações citadas na seção 2.5.1, define-se que devem ser observadas as heurísticas de minimização de cruzamentos, dobras e comprimento de arestas.

As etapas principais do algoritmo, listadas na figura 5.7, são semelhantes às do algoritmo do editor Smart. O algoritmo inicia com a detecção e eliminação de ciclos com a inversão de uma aresta de cada ciclo encontrado. Na etapa de hierarquização os vértices

do grafo são particionados em camadas, que são então ordenadas de acordo com as heurísticas definidas. Na etapa de normalização, vértices fictícios são criados de forma que todas asa arestas conectam vértices de camadas adjacentes. A etapa de redução de cruzamentos é a mais próxima do algoritmo de Sugiyama original: os vértices de cada camada são ordenadas de acordo com o baricentro de seus vizinhos, iterativamente. Na etapa de layout métrico, são calculadas as dimensões e posições de cada vértice, considerando também as heurísticas definidas.

1. Eliminação de ciclos
2. Hierarquização
   (a) Criação de camadas
   (b) Ordenação de camadas
3. Normalização
   (a) Compactação de camadas
   (b) Criação de bordas de subgrafos
   (c) Criação de vértices intermediários
4. Redução de cruzamentos
5. Layout métrico
   (a) Criação de vértices intra-ciclos
   (b) Atribuição de prioridades
   (c) Posicionamento de vértices

Figura 5.7: Algoritmo de layout proposto

5.4.3.1 Hierarquização

O algoritmo do editor Smart cria uma camada a partir dos níveis da árvore de inclusão do grafo. Esse método de hierarquização não é adequado ao algoritmo do editor Sins porque ignora as arestas do dígrafo de adjacências, impossibilitando a observação das heurísticas definidas.

Para superar essa limitação, a hierarquização neste algoritmo cria camadas a partir dos vértices do grafo. Cada subgrafo dá origem a dois vértices: um marcado como − (menos)

e outro marcado como + (mais), correspondendo respectivamente às bordas horizontais superior e inferior do subgrafo. Cada vértice base (folha da árvore de estados) dá origem a um vértice. Os rótulos das transições dão origem a um vértice cada. Todos os vértices criados na fase de hierarquização são colocados em camadas que contêm apenas um vértice cada, como mostrado na figura 5.8.

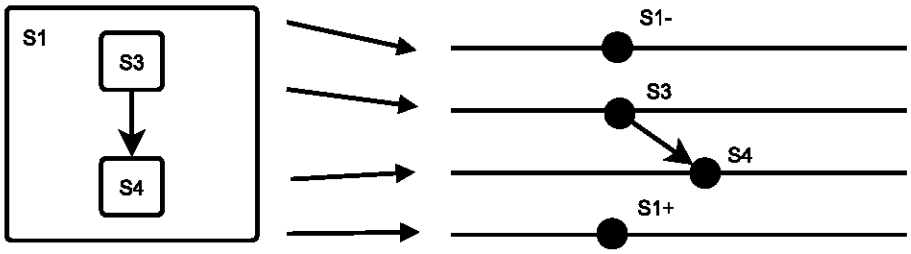


Figura 5.8: Etapa de hierarquização

Dado um dígrafo composto $D = (V, E, F)$, define-se o conjunto de subgrafos de D como $S = \{u : uv \in E\}$ e o conjunto de folhas ou vértices base de D como $B = V - S$. O conjunto de pares de vértices correspondentes aos limites superior e inferior dos subgrafos no mapa são definidos respectivamente como $S^- = \{v^- : v \in S\}$ e $S^+ = \{v^+ : v \in S\}$. Para fins de concisão na notação, $S' = S^- \cup S^+$ e $V' = B \cup S'$.

Seja a função An tal que $An(u)$ é o conjunto dos vértices ancestrais de u, ou seja, existe um caminho em E entre qualquer vértice de $An(u)$ e u. Uma hierarquização é definida como uma função injetora $H : V' \rightarrow \mathbb{N}$ que atende às seguintes restrições, relativas a quantidade de vértices em cada camada e à manutenção das relações de inclusão do dígrafo composto:

$$\forall u, v \in V'; H(u) = H(v) \leftrightarrow u = v$$

$$\forall (u, v) \in S \times V, u \in An(v) \rightarrow H\left(u^-\right) < H(v) < H\left(u^+\right)$$

Para atender às métricas de minimizar o comprimento das arestas, as camadas são ordenadas de forma a posicionar vértices conexos próximos uns aos outros. Isso é feito através da ordenação dos vértices de acordo com o valor dos seus baricentros, de maneira similar à redução de cruzamentos do algoritmo de Sugiyama. Também são garantidas, nessa ordem:

- a manutenção das relações de inclusão de vértices (posicionando vértices de subgrafos "menos" e "mais" antes e depois, respectivamente, dos seus descendentes exclusivamente) e

- o fluxo de conexões entre vértices, posicionando vértices de origem antes de vértices de destino (quando conexos).

Assim, a intersecção vertical de subgrafos não relacionados é evitada. A ordenação de camadas é ilustrada na figura 5.9 e descrita a seguir.

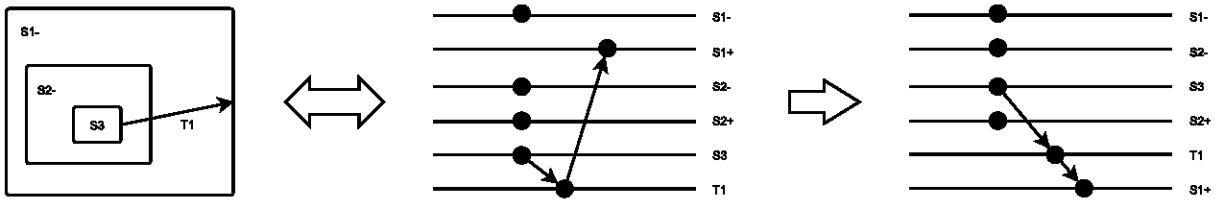


Figura 5.9: Ordenação de camadas

O conjunto $\mathcal{H}$ é o conjunto de todas as hierarquizações existentes para o dígrafo D. A função $N_H : V^{'} \times S \times \mathcal{H} \rightarrow V^{'}$ recebe como parâmetros vértices $v \in V^{'}$ e $s \in S$ e uma hierarquização H e retorna o vértice fictício correspondente a s mais próximo de v em H.

$$N_H(v,s,H) = \begin{cases} s^- & \leftarrow & |H(v) - H(s^-)| < |H(v) - H(s^+)| \\ s^+ & \leftarrow & |H(v) - H(s^-)| \geq |H(v) - H(s^+)| \end{cases}$$

A partir de N_H, podemos definir as funções $N_V : V \times S \times \mathcal{H}$, que retorna o vértice fictício correspondente a $s \in S$ mais próximo de $v \in V$, e $N_S : S \times S \times \mathcal{H}$, que retorna o vértice fictício corresponde a s mais próximo de um dos vértices fictícios de $v \in S$. Sendo $h_N^- = H(N_H(v^-, s, H))$ e $h_N^+ = H(N_H(v^+, s, H))$, N_V e N_S correspondem a:

$$N_V(v,s,H) = N_H(H(v), s, H)$$

$$N_S(v,s,H) = \begin{cases} N_H(v^-,s,H) & \leftarrow & \left|H(v^-) - h_N^-\right| < \left|H(v^+) - h_N^+\right| \\ N_H(v^+,s,H) & \leftarrow & \left|H(v^-) - h_N^-\right| \geq \left|H(v^+) - h_N^+\right| \end{cases}$$

Seja o conjunto $F^{'}(H) = F_1 \cup F_2(H) \cup F_3(H) \cup F_4(H)$ de todas as arestas de adjacência em F mapeadas para arestas em $V^{'} \times V^{'}$ segundo $H \in \mathcal{H}$. Arestas entre vértices pertencentes a S são mapeadas para os vértices fictícios mais próximos. Assumindo $uv \in F$, definem-se os componentes de $F^{'}$ como:

$$\begin{aligned} F_1 &= \{uv : u, v \in B\} \\ F_2(H) &= \left\{uv^{'} : (u,v) \in B \times S \wedge v^{'} = N_V(u,v,H)\right\} \\ F_3(H) &= \left\{u^{'}v : (u,v) \in S \times B \wedge u^{'} = N_V(v,u,H)\right\} \\ F_4(H) &= \left\{u^{'}v^{'} : u, v \in S \wedge u^{'} = N_S(u,v,H) \wedge v^{'} = N_S\left(v, u^{'}, H\right)\right\} \end{aligned}$$

Seja $n(v,H) = \left\{u : uv \in F^{'}(H) \vee vu \in F^{'}(H)\right\}$ o conjunto de todos os vizinhos do vértice $v \in V$ na hierarquização H. A função $\beta : V^{'} \times \mathcal{H} \rightarrow \mathbb{R}$ retorna o baricentro dos vizinhos de um vértice $v \in V^{'}$ em $H \in \mathcal{H}$ como definido a seguir:

$$\beta(v,H) = \begin{cases} \sum_{x \in n(v,H)} \frac{H(x)}{|n(v,\Pi)|} & \leftarrow \quad |n(v,\Pi)| > 0 \\ \Pi(v) & \leftarrow \quad |n(v,\Pi)| = 0 \end{cases}$$

Seja $D_I \subset V^{'}$ o conjunto de estados iniciais do grafo e $D_F \subset V^{'}$ o conjunto de estados finais, de forma que $D_I \cap D_F = \emptyset$. A etapa de hierarquização pode ser descrita por uma seqüência de hierarquizações $H_1 H_2 ... H_L$ de tal forma que $H_i \in \mathcal{H}$, L é o limite máximo de iterações e:

$$\forall u, v \in V^{'}; \beta(u, H_i) < \beta(v, H_i) \rightarrow H_{i+i}(u) < H_{i+i}(v)$$

$$\forall (u,v) \in D_I \times V^{'}, uv \in F^{'}(\Pi_i) \rightarrow \Pi_{i+1}(u) < \Pi_{i+1}(v)$$

$$\forall (u,v) \in V^{'} \times D_F, uv \in F^{'}(H_i) \rightarrow H_{i+1}(u) < H_{i+1}(v)$$

Dessa forma, os vértices são organizados de maneira a minimizar o seu comprimento, a posicionar estados iniciais acima de seus vizinhos e estados finais abaixo deles.

5.4.3.2 Normalização

Na fase de normalização as camadas, que contêm apenas um vértice cada, são compactadas quando isso não contraria as necessidades do algoritmo (vértices conexos ou com relação de inclusão na hierarquia de estados devem ficar em camadas diferentes). Apenas camadas sucessivas são compactadas; a compactação de camadas é mostrada na figura 5.10.

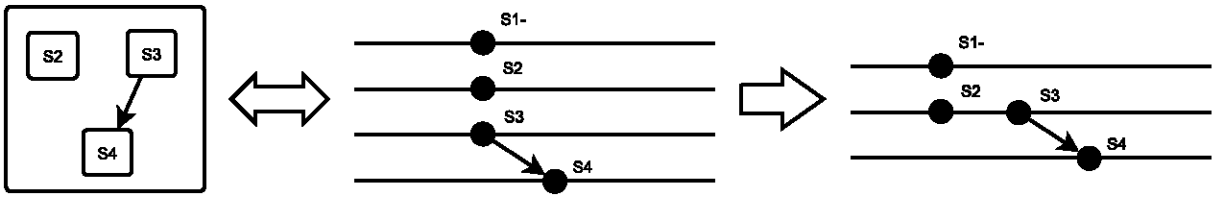


Figura 5.10: Compactação de camadas

Pode-se criar uma hierarquização compactada $N_C : V^{'} \rightarrow \mathbb{N}$ derivada de Π_L tal que a ordem das camadas é mantida e camadas sucessivas que representam vértices base e desconexos são unidas em uma única camada. Essas restrições podem ser descritas pelas equações:

$$\forall u, v \in V^{'}; H_L(u) < H_L(v) \rightarrow N_C(u) \leq N_C(v)$$

$$\forall u, v \in B; uv \notin F \wedge (H_L(u) = H_L(v) - 1) \rightarrow N_C(u) = N_C(v)$$

Neste algoritmo é importante evitar que subgrafos sejam posicionados de forma a existir intersecção entre eles, a não ser quando existe uma relação de inclusão e um subgrafo é totalmente contido no outro. Para evitar a intersecção horizontal de subgrafos, são criados vértices de bordas de subgrafos, responsáveis por conter os descendentes do subgrafo. Para cada vértice de subgrafo marcado como "menos" são criados dois vértices marcados simultaneamente como "menos" e L (esquerda) ou R (direita). O equivalente acontece com os vértices de subgrafos marcados como "mais". Os vértices L "menos" e "mais" de um mesmo subgrafo são ligados por uma aresta que conecta o primeiro ao segundo; o equivalente acontece com os vértices marcados como R. Esse procedimento é ilustrado na figura 5.11.

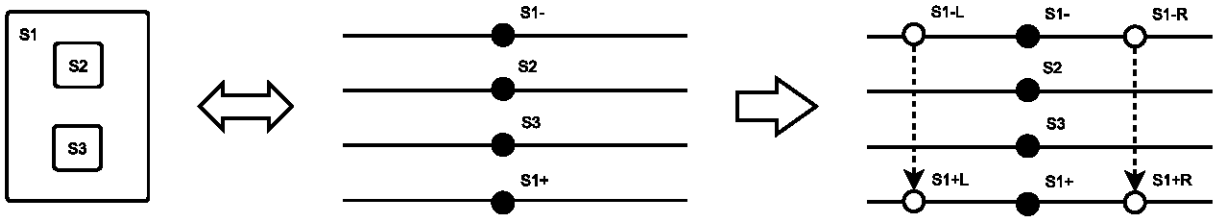


Figura 5.11: Criação de bordas de subgrafos

Para a criação das bordas dos subgrafos, são definidos os conjuntos de vértices $S_B = \left\{s_L^-, s_R^-, s_L^+, s_R^+ : s \in S\right\}$ e $V_B = \left(V_F - S^{'}\right) \cup S_B$. Seja $N_B : V_B \rightarrow \mathbb{N}$ tal que:

$$\forall s \in S^{'}, N_F(s) = N_B(s_L) = N_B(s_R)$$

$$\forall x \notin S^{'}, N_F(x) = N_B(x)$$

Sejam as funções $p_{SL} : S \rightarrow V_B^+$ e $p_{SR} : S \rightarrow V_B^+$ tal que $p_{SL}(x)$ retorna o caminho entre x_L^- e x_L^+ e $p_{SR}(x)$ retorna o caminho entre x_R^- e x_R^+. O novo conjunto de arestas correspondente a $F_B = F_N \cup \{uv : uv \in p_{SL}(s \in S)\} \cup \{uv : uv \in p_{SR}(s \in S)\}$.

Para a fase de criação de vértices intermediários define-se V_f o conjunto de vértices intermediários fictícios, $V_N = V^{'} \cup V_f$ e a função $p_N : V^{'} \times V^{'} \rightarrow (V_N)^+$ que retorna um caminho entre dois vértices de camadas diferentes:

$$p_N(u,v) = \begin{cases} uv & \leftarrow & |N_C(u) - N_C(v)| = 1 \\ ux_2...x_{n-1}v & \leftarrow & |N_C(u) - N_C(v)| > 1 \end{cases}, x_i \in V_f$$

Note-se que todos os caminhos retornados por p_N são disjuntos, isto é:

$$\forall u,v,w,x \in V^{'}; \exists x \in p_N(u,v); x \in p_N(w,x) \leftrightarrow u = w \wedge v = x$$

Como no algoritmo de Sugiyama, as camadas devem ser normalizadas. Isso significa que arestas que conectam vértices de camadas que não são sucessivas precisam ser quebradas em uma sequência de vértices fantasma em cada camada intermediária e arestas que mantêm a conexão. Isso é mostrado na figura 5.12.

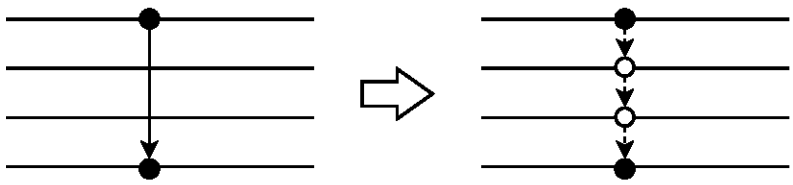


Figura 5.12: Criação de vértices intermediários

O conjunto de arestas que inclui aquelas entre vértices intermediários corresponde a

$$F_N = \bigcup_{uv \in F^{'}(N_C)} \{x_i x_{i+1} \in p(u,v) : 1 \leq i \leq |p(u,v)|\}$$

A hierarquização normalizada é definida pela função $N_F : V_N \rightarrow \mathbb{N}$ derivada de N_C tal que:

$$\forall uv \notin F^{'}(N_C), N_F(u) = N_C(u) \wedge N_F(v) = N_C(v)$$

$$\forall uv \in F^{'}(N_C), x_1x_2...x_n = p(u,v) \wedge N_F(x_i) = N_C(x_1) + i - 1$$

Observa-se que $\forall uv \in F_N, |N_F(u) - N_F(v)| = 1$, caracterizando a normalização das camadas.

5.4.3.3 Redução de cruzamentos

Uma das métricas mais importantes para a legibilidade de um diagrama é a minimização de cruzamentos entre arestas. Isso é conseguido com a ordenação iterativa dos vértices de cada camada de acordo com os seus baricentros. Semelhante ao algoritmo de Sugiyama, esse processo é mostrado na figura 5.13.

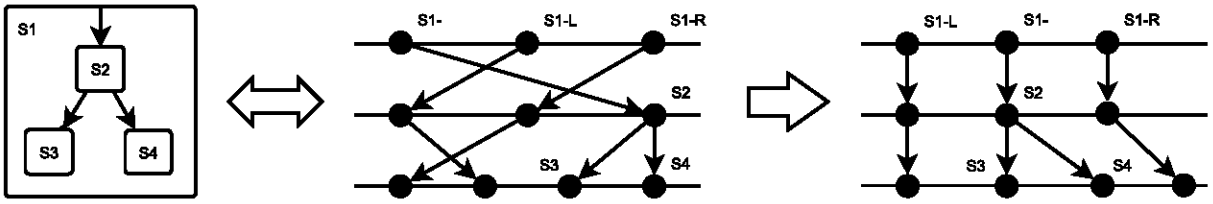


Figura 5.13: Redução de cruzamentos

Sejam $l(v) = \{x : N_B(x) = N_B(v)\}$ o conjunto de vértices na mesma camada de v (inclusive ele próprio) e $L = \{l(v) : v \in V_B\}$ o particionamento dos vértices de V_B em suas camadas. A função $t(l) = N_B(x), x \in l$ retorna a posição da camada $l \in L$ de acordo com a ordem definida pela hierarquização N_B.

Define-se $\mathcal{K}$ como o conjunto de todas as funções de ordenação injetoras $k : V_B \to \mathbb{N}$ tal que sendo $l \in L$:

$$\forall u, v \in l; k(u) = k(v) \leftrightarrow u = v$$

$$\forall (u, v) \in S \times V, u \in An(v) \leftrightarrow k(u_L) < k(v) < k(u_R)$$

Assim, cada vértice $v \in V_B$ tem sua posição dentro de sua camada dada por $k \in \mathcal{K}$.

O conjunto de vizinhos da camada seguinte àquela do vértice $v \in V_B$ é dado por $n_d(v) = \{x : N_B(x) = N_B(v) + 1 \wedge (vx \in F_B \vee xv \in F_B)\}$. Analogamente, o conjunto de vizinhos da camada anterior a v é dado por $n_u(v) = \{x : N_F(x) = N_F(v) - 1 \wedge (vx \in F_B \vee xv \in F_B)\}$ Sejam as funções β_d e β_u, ambas sobre $V_B \times \mathcal{K} \to \mathbb{R}$ e retornando o baricentro dos vizinhos das camadas seguinte e anterior, respectivamente, definidas como:

$$\beta_d(v, k) = \begin{cases} \sum_{x \in n_d(v)} \frac{k(x)}{|n_d(v)|} & \leftarrow \quad |n_d(v)| > 0 \\ N_B(v) & \leftarrow \quad |n_d(v)| = 0 \end{cases}$$

$$\beta_u(v, k) = \begin{cases} \sum_{x \in n_u(v)} \frac{N_B(x)}{|n_u(v)|} & \leftarrow \quad |n_u(v)| > 0 \\ N_B(v) & \leftarrow \quad |n_u(v)| = 0 \end{cases}$$

As iterações da fase descendente da redução de cruzamentos são descritas pela seqüência $d_1 d_2 ... d_{n-1}$ onde $d_i \in \mathcal{K}$, $n = |L|$, $l_i \in L$, $\forall x \in l_i, N_B(x) = i$ e:

$$\forall u, v \in l_i; \beta_d(u, d_i) < \beta_d(v, d_i) \rightarrow d_{i+1}(u) < d_{i+1}(v)$$

As iterações da fase ascendente dessa etapa são descritas pela seqüência $a_n a_{n-1} ... a_2$ onde $a_i \in \mathcal{K}$ e:

$$\forall u, v \in l_i; \beta_u(u, a_i) < \beta_u(v, a_i) \rightarrow a_{i-1}(u) < a_{i-1}(v)$$

As iterações das fases descendente e ascendente dessa etapa se repetem até que o número máximo de iterações L seja atingido, obtendo assim um número menor de cruzamentos entre arestas dado pela ordenação $k_M \in \mathcal{K}$.

5.4.3.4 Layout métrico

Os ciclos, neste algoritmo, são mostrados em uma disposição aproximadamente circular. Para isso, vértices fantasma são criados conectando o primeiro e o último vértices de um ciclo (definidos como o vértice pertencente ao ciclo posicionado na camada mais acima e na camada mais abaixo, respectivamente). Um vértice é criado para cada camada intermediária e arestas conectam esses vértices. Isso é descrito a seguir e ilustrado na figura 5.14.

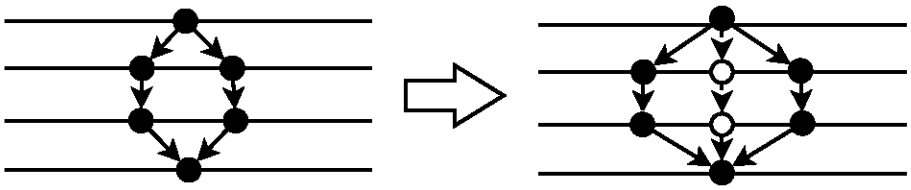


Figura 5.14: Criação de vértices intra-ciclos

Sendo C o conjunto de todos os ciclos no grafo $D = (V_B, F_B)$. É definida a função $p_C : C \rightarrow V_B$ tal que $p_C(c) = c_{min} i_1 i_2 ... i_n c_{max}$ é o caminho entre o vértice c_{min} de nível mínimo dado por $N_B(c_{min})$ e o vértice c_{max} de nível máximo dado por $N_B(c_{max})$; os vértices $i_1 ... i_n$ pertencem ao conjunto V_I de vértices intra-ciclos. O novo conjunto de arestas é definido como $F_I = F_B \cup \{uv : uv \in p_C(c), c \in C \wedge |p_C(c)| > 2\}$.

Vértices de bordas de subgrafos têm a maior prioridade na fase de posicionamento de vértices, seguidos pelos intra-ciclos, pelos vértices intermediários resultantes da segmentação de arestas que ligam vértices de camadas não sucessivas e pelos outros vértices. A cada vértice $v \in V_B$ é atribuída uma prioridade $P(v)$:

$$P(v) = \begin{cases} n(v) & \leftarrow & v \in V' \\ P_f & \leftarrow & v \in V_f \\ |p_C(c)| & \leftarrow & v \in V_I, v \in p_C(c), c \in C \\ P_B & \leftarrow & v \in V_B \end{cases}$$

Para posicionar os vértices horizontalmente, estes são posicionados em colunas e movidos iterativamente de forma a se aproximar dos seus vizinhos, similarmente ao algoritmo de Sugiyama. Esse processo é mostrado na figura 5.15.

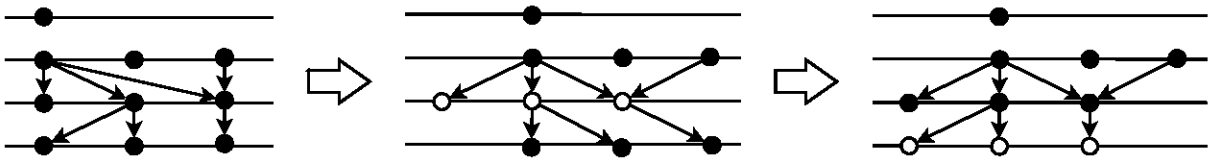


Figura 5.15: Posicionamento de vértices

Define-se o baricentro β_M dos vizinhos de $v \in V_B$ como:

$$\beta_M(v) = \frac{\beta_d(v, k_M) + \beta_u(v, k_M)}{2}$$

A posição horizontal de cada vértice é dada pela função $T : V_B \to \mathbb{R}$ tal que:

$$\forall u, v \in V_B, T(u) < T(v) \leftrightarrow k_M(u) < k_M(v)$$

$$\forall u, v \in V_B, P(u) > P(v) \to |T(u) - \beta_M(u)| \leq |T(v) - \beta_M(v)|$$

Assim, vértices intra-ciclos, intermediários e de borda são, nessa ordem, posicionados o mais próximo possível do baricentro médio de seus vizinhos. Dessa forma, ciclos são alinhados horizontalmente em relação ao seu centro, dando priorizando o alinhamento de ciclos maiores em detrimento de ciclos menores; arestas que atravessam mais de uma camada têm seu número de dobras diminuído e as bordas de um mesmo estado são alinhadas verticalmente.

5.5 Conclusão

Para a implementação do editor Sins, foram considerados, além dos requisitos funcionais, requisitos de sistema como independência de plataforma, licença compatível com código livre e extensibilidade. Entre as tecnologias que atendem a esses requisitos, foram selecionados Java como linguagem e plataforma de implementação, Eclipse como ambiente de desenvolvimento extensível, o framework GEF para implementação de editor gráfico dentro do Eclipse e o JavaCC como gerador de parsers.

A implementação do editor foi organizada em componentes que seguem o padrão de projeto MVC. O controle é responsabilidade do editor de especificação, as visões são fornecidas por componentes correspondentes a editores Xchart, de eventos, de atividades, de portas e TEXchart e o modelo equivale à especificação, sendo acessado por intermédio de um gerente de conteúdo. O gerente de conteúdo é responsável pela sincronização e tradução entre o modelo e sua representação textual na linguagem TEXchart.

O algoritmo de layout automático implementado é similar ao algoritmo do editor Smart, que por sua vez é uma variação do algoritmo de Sugiyama para dígrafos compostos. O algoritmo implementado utiliza métodos diferentes na etapa de hierarquização, de normalização e de layout métrico com o objetivo de atender melhor às heurísticas definidas. Assim, o mapa gerado apresenta as estruturas do diagrama de forma consistente, com poucos cruzamentos entre arestas, poucas dobras em arestas, vértices próximos de seus vizinhos, boa simetria e se aproxima dos mapas criados manualmente pelos usuários.

Capítulo 6

Palavras Finais

O desenvolvimento de sistemas reativos através de uma arquitetura orientada a modelos contribui significativamente para diminuir o tempo entre o seu projeto e a sua implementação ao possibilitar a utilização de ferramentas que automatizam partes desse processo. O uso dessas ferramentas também aumenta a confiabilidade dos sistemas criados por tomar do desenvolvedor a responsabilidade pela implementação de estruturas repetitivas e detalhes dependentes de sistema.

A linguagem Xchart, uma das disponíveis para a modelagem de sistemas reativos, é uma linguagem visual voltada à criação de diagramas, mas as ferramentas do ambiente de desenvolvimento Xchart interpretam sua equivalente textual TEXchart. A distância entre a linguagem visual que serve de base para a modelagem dos sistemas e a linguagem textual que é compreendida pelas ferramentas de desenvolvimento prejudica os processos de projeto e depuração desses sistemas. Assim, é interessante a utilização de um editor de diagramas Xchart que traduza transparentemente entre as várias representações possíveis de uma especificação.

As duas principais contribuições deste trabalho são a implementação do editor Sins e a definição do algoritmo de layout automático usado pelo editor. O editor Sins supera algumas das limitações do editor Xchart já existente (Smart). Entre as vantagens do Sins, destacam-se:

- a edição de diagramas através de manipulação direta,
- a tradução transparente entre as linguagens Xchart e TEXchart,
- a implementação de um algoritmo de layout automático mais adequado a diagramas Xchart,
- a independência da plataforma Microsoft Windows e

- integração com o ambiente de desenvolvimento integrado Eclipse.

O algoritmo de layout automático do editor Sins gera layouts consistentes na apresentação das estruturas do diagrama, além de usar um estilo (regras de posicionamento dos elementos de um tipo de diagrama) mais próximo ao dos diagramas desenhados manualmente. Além dessas características, métricas como a minimização de cruzamentos, dobras e comprimento de arestas orientam o algoritmo e contribuem para a legibilidade dos layouts gerados.

Dentre as possibilidades de trabalhos futuros, podemos destacar melhorias no editor Sins:

- a integração com o compilador TEXchart com o ambiente de desenvolvimento, permitindo a geração automatizada e transparente de código executável equivalente às especificações editadas e
- a integração entre as facilidades de depuração oferecidas pelo ambiente Eclipse e o sistema de execução Xchart, oferecendo uma simulação visual das especificações editadas e diminuindo significativamente a duração do ciclo projeto-implementação-depuração.

Outra direção para melhorias é o algoritmo de layout, que não considera o mapa mental do usuário. Tal característica minimizaria o efeito, no mapa gerado pelo algoritmo, das mudanças estruturais introduzidas pelo usuário durante a edição. Considerar tal efeito no algoritmo diminuiria o esforço cognitivo do usuário ao reinterpretar mudanças na disposição dos elementos do diagrama e aumentaria sensivelmente a usabilidade do editor. Uma alternativa é a possibilidade de bloquear a posição de determinados elementos do diagrama, aplicando o algoritmo no restante do mesmo.

Referências Bibliográficas

[1] HAREL, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, jun 1987.

[2] UML, O. Uml specification version 1.3. *Object Management Group*, 1999.

[3] BENVENISTE, A.; BERRY, G. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, v. 79, n. 9, p. 1270–1282, 1991.

[4] ROSCOE, A. W. *The Theory and Practice of Concurrency*. [S.l.]: Prentice-Hall (Pearson), 2005.

[5] LUCENA, F. N. de; LIESENBERG, H. K. E. *Interfaces Homem-Computador: Uma Primeira Introdução*. [S.l.], set. 1994. 29 p.

[6] CHAU, L. H.; CHAN, G. K. Visual language for behavioral specifications of reactive systems. In: *International Conference on Computer Languages*. [S.l.: s.n.], 1994.

[7] SHNEIDERMAN, B. Multiparty grammars and related features for defining interactive systems. *IEEE Transactions on Systems, Man, and Cybernetics*, v. 12, n. 2, p. 148–154, 1982.

[8] JACOB, R. Using formal specifications in the design of a human-computer interface. *Communications of the ACM*, ACM Press New York, NY, USA, v. 26, n. 4, p. 259–264, 1983.

[9] MURATA, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, v. 77, n. 4, p. 541–580, 1989.

[10] DAVID, R.; ALLA, H. *Petri nets and Grafcet*. [S.l.]: Prentice Hall, 1992.

[11] IEC. *Standard 61131-3: Programmable controllers Part 3*. IEC, 2003. Disponível em: <http://webstore.iec.ch/webstore/webstore.nsf/artnum/029664>.

[12] MARANINCHI, F. *The Argos language: Graphical representation of automata and description of reactive systems.* out 1991. IEEE Workshop on Visual Languages. Disponível em: <citeseer.ist.psu.edu/maraninchi91argo.html>.

[13] LUCENA, F. Nogueira de; LIESENBERG, H. K. E.; BUZATO, L. E. *Especificação da Linguagem Xchart.* Campinas, São Paulo.

[14] ANDRé, C. Representation and analysis of reactive behaviors: A synchronous approach. In: *Computational Engineering in Systems Applications.* [S.l.: s.n.], 1996. p. 19–29.

[15] AUBURN, R. et al. *State Chart XML (SCXML): State Machine Notation for Control Abstraction 1.0.* [S.l.], jul. 2005.

[16] BAETEN, J. A brief history of process algebra. *Theoretical Computer Science*, v. 335, n. 2, p. 131–146, 2005.

[17] BHARADWAJ, R. Sol: A verifiable synchronous language for reactive systems. *Electr. Notes Theor. Comput. Sci.*, v. 65, n. 5, 2002.

[18] GROUP, R. S. *Quartz Language Reference Manual.* 1.8. ed. Alemanha, mai 2006.

[19] CAMPOS, J. C. *GAMA-X: Geração Semi-Automática de Interfaces Sensíveis ao Contexto.* Dissertação (Mestrado) Departamento de Informática, Universidade do Minho, 1993. Disponível em: <http://www.di.uminho.pt/ jfc/research/techrep/Campos93/node10.html>.

[20] GREEN, M. Report on dialogue specification tools. *User Interface Management Systems*, 1985.

[21] REYNOLDS, C. A critical examination of separable user interface management systems: Constructs for individualization. *SIGCHI Bulletin*, v. 29, n. 3, jul 1997.

[22] TECNOLOGIA Xchart. aug 2005. Disponível em: <http://www.dcc.unicamp.br/proj-xchart/geral/geral.html>.

[23] BROWN, A. An introduction to model driven architecture part i: Mda and today's systems. *The Rational Edge*, fev 2004. Disponível em: <http://www-128.ibm.com/developerworks/rational/library/3100.html>.

[24] KONTIO, M. Architectural manifesto: Mda for the enterprise. *developerWorks*, jul 2005. Disponível em: <http://www-128.ibm.com/developerworks/library/wi-arch16/index.html>.

[25] MICROSYSTEMS, S. Java server pages. mai. 2004.

[26] CORPORATION, M. *Active Server Pages Tutorial.* dez. 2000. Active Server Pages Technical Articles. Disponível em: <http://msdn2.microsoft.com/en-us/library/ms972337.aspx>.

[27] GROUP, O. M. *Unified Modeling Language: Superstructure.* 2.1.1. ed. [S.l.], fev 2007.

[28] CHOK, S. S.; MARRIOTT, K. Automatic generation of intelligent diagram editors. *ACM Transactions on Computer-Human Interaction*, v. 10, n. 3, p. 244–276, set 2003.

[29] NORTH, S. C.; KOUTSOFIOS, E. *Applications of Graph Visualization.* Murray Hill, New Jersey.

[30] LUCAS, P. J. *A Graphical Editor Proposal for Developing Concurrent, Hierarchical, Finite State Machines.* Urbana, Illinois, jan 1993.

[31] CASTELLÓ, R.; MILI, R.; TOLLIS, I. G. *Visualizing Statecharts with ViSta.* Chihuahua, México.

[32] CASTELLÓ, R.; MILI, R.; TOLLIS, I. G. A framework for the static and interactive visualization of statecharts. *Journal of Graph Algorithms and Applications*, Dallas, v. 6, n. 3, p. 313–351, 2002.

[33] SUGIYAMA, K.; MISUE, K. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Software Engineering*, v. 21, n. 4, p. 876 892, 1991.

[34] JÚNIOR, O. S. *Smart: um Editor Gráfico de Diagramas em Xchart.* Dissertação (Mestrado) Instituto de Computação, Universidade Estadual de Campinas, Campinas, São Paulo, 1996.

[35] EICHELBERGER, H.; GUDENBERG, J. Wolff von. *Demonstration of Advanced Layout of UML Class Diagrams by SugiBib.* Alemanha.

[36] IBM CORPORATION E OUTROS. *GEF and Draw2d Plug-in Developer Guide.* [S.l.], 2005.

[37] MINAS, M.; HOFFMANN, B. Specifying and implementing visual process modeling languages with diagen. *Electronic Notes in Theoretical Computer Science*, v. 44, n. 4, 2001.

[38] MINAS, M. Specifying graph-like diagrams with diagen. *Electronic Notes in Theoretical Computer Science*, v. 72, n. 2, p. 10, 2002.

[39] EHRIG, K. et al. *Generation of Visual Editors as Eclipse Plug-Ins.* [S.l.], 2005.

[40] TAENTZER, G. et al. *Tiger Project.* 1.2.0. ed. [S.l.].

[41] PURCHASE, H. C. Performance of layout algorithms: Comprehension, not computation. *Journal of Visual Languages and Computing*, Academic Press, v. 9, n. 6, p. 647–657, 1998.

[42] PURCHASE, H. et al. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. *ACM International Conference Proceeding Series*, Australian Computer Society, Inc. Darlinghurst, Australia, Australia, p. 129–137, 2001.

[43] PURCHASE, H.; ALLDER, J.; CARRINGTON, D. Graph layout aesthetics in uml diagrams: User preferences. *Journal of Graph Algorithms and Applications*, v. 6, n. 3, p. 255 279, 2002.

[44] FRIEDRICH, C.; EADES, P. Graph drawing in motion. *Journal of Graph Algorithms and Applications*, v. 6, n. 3, p. 353–370, 2002.

[45] NORTH, S. C. *Incremental Layout in DynaDAG.* [S.l.].

[46] KONING, H.; DORMANN, C.; VLIET, H. van. Practical guidelines for the readability of it-architecture diagrams. In: *SIGDOC'02.* [S.l.: s.n.], 2002.

[47] MISUE, K. et al. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, v. 6, p. 183–210, 1995.

[48] EADES, P.; HUANG, M. L. Navigating clustered graphs using force-directed methods. *Journal of Graph Algorithms and Applications*, v. 4, n. 3, p. 157–181, 2000.

[49] ENRIGHT, J. A.; OUZOUNIS, C. A. Biolayout an automatic graph layout algorithm for similarity visualization. *Bioinformatics Applications Note*, v. 17, n. 9, p. 853 854, 2001.

[50] FRÖHLICH, M.; WERNER, M. *The Graph Visualization System daVinci – A User Interface for Applications.* Bremen, Alemanha, 1994.

[51] JUUSTISTENAHO, A. *Linear Time Algorithms for Layout of Generalized Trees.* Finlândia, 1994.

[52] GANSNER, E. R. et al. *Graph Visualization in Software Analysis.* [S.l.].

[53] TATZMANN, B. *Dynamic Exploration of Large Graphs.* Dissertação (Mestrado) — Universidade de Tecnologia Graz, Graz, Áustria, mar 2004.

[54] EADES, P.; FENG, Q. *Multilevel Visualization of Clustered Graphs.* Austrália.

[55] SUGIYAMA, K.; TAGAWA, S.; TODA, M. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on System, Man and Cybernetics*, v. 11, n. 2, p. 1047 1062, 1981.

[56] SHNEIDERMAN, B. Direct manipulation: A step beyond programming languages. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, p. 461 467, 1987.

[57] EDGAR, N. et al. *Eclipse User Interface Guidelines Version 2.1.* fev 2004. Disponível em: <http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html>.

[58] DAM, A. van. Post-wimp user interfaces. *Communications of the ACM*, v. 40, n. 2, p. 63 67, fev. 1997.

[59] BECK, K.; CUNNINGHAM, W. *Using Pattern Languages for Object-Oriented Programs.* [S.l.], set. 1987. Disponível em: <http://c2.com/doc/oopsla87.html>.

[60] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* [S.l.]: Addison-Wesley Professional, 1995.

[61] CAI, J.; KAPILA, R.; PAL, G. *HMVC: The layered pattern for developing strong client tiers.* jul. 2000. JavaWorld.com. Disponível em: <http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.html?page=1>.

[62] JÚNIOR, C. N. *Gerente de Distribuição do Ambiente Xchart em J2EE.* Dissertação (Mestrado) Instituto de Computação, Universidade Estadual de Campinas, Campinas, São Paulo, set 2005.

[63] TECNOLOGIA Xchart: Formato de Arquivos. ago 2006. Disponível em: <http://www.dcc.unicamp.br/proj-xchart/texchart/formato.html>.

[64] SMITH, J. et al. Achieving high performance via co-designed virtual machines. *Innovative Architecture for Future Generation High-Performance Processors and Systems, 1998*, p. 77 84, 1999.

[65] TROLLTECH. *Qt 4.2 Whitepaper.* 2006.

[66] SMART, J. et al. *wxWidgets 2.8.3: A portable C++ and Python GUI toolkit.* [S.l.], mar 2007.

[67] ORACLE. *Oracle JDeveloper Overview.* jan 2007.

[68] HAASTER, K. V.; HAGAN, D. Teaching and learning with bluej: an evaluation of a pedagogical tool. *Issues in Informing Science and Information Technology*, p. 455 470, jun 2004.

[69] SUN. *Using NetBeans IDE 5.5*. [S.l.].

[70] SCHREINER, A. T. *Package jay Description*. jun 2006. Disponível em: <http://www.cs.rit.edu/ ats/projects/lp/doc/jay/package-summary.html>.

[71] BEAVER – a LALR Parser Generator. mai 2007. Disponível em: <http://beaver.sourceforge.net/>.

[72] HURKA, T. *BYACC/J Home Page*. mai 2007. Disponível em: <http://byaccj.sourceforge.net/>.

[73] HUDSON, S. E. et al. *CUP User's Manual*. [S.l.], jul 1999.

[74] ANTLR Reference Manual. [S.l.].

[75] MöSSENBöCK, H. *The Compiler Generator Coco/R*. [S.l.], 2006.

[76] GAGNON Étienne. *SableCC, An Object-Oriented Compiler Framework*. Dissertação (Mestrado) — Escola de Ciência da Computação, Universidade McGill, Montreal, mar 1998.

[77] FOX, G.; FURMANSKI, W. Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency Practice and Experience*, v. 9, n. 6, p. 415 425, 1997.

[78] BULL, J. M. et al. Benchmarking java against c and fortran for scientific applications. *Concurrency and Computation: Practice & Experience*, v. 15, n. 3, p. 417 430, 2003.

[79] KRAMER, D.; JOY, B.; SPENHOFF, D. *The Java[tm] Platform*. [S.l.], 1996. Disponível em: <http://java.sun.com/docs/white/platform/javaplatformTOC.doc.html>.

[80] THE OSGI ALLIANCE. *OSGi Service Platform Core Specification*. [S.l.], jul. 2006.

[81] IBM CORPORATION E OUTROS. *Platform Plug-in Developer Guide*. [S.l.], 2006.

[82] JAVACC Documentation. mai 2007. Incluído no código fonte do programa.

[83] TECNOLOGIA Xchart: Conceitos Através de Exemplos. 1997.

[84] NIELSEN, J. *Usability inspection methods*. [S.l.]: ACM Press New York, NY, USA, 1995.

Índice Remissivo