Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Uma Metodologia para Especificar Interação 3D utilizando Redes de Petri

Rafael Rieder

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Márcio Serolli Pinho

Porto Alegre
2007

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Uma Metodologia para Especificar Interação 3D Usando Redes de Petri***", apresentada por Rafael Rieder, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Computação Cientifica, aprovada em 21/12/2006 pela Comissão Examinadora:

Prof. Dr. Márcio Serolli Pinho – Orientador — PPGCC/PUCRS

Prof. Dr. João Batista Souza de Oliveira — PPGCC/PUCRS

Prof. Dr. Alberto Barbosa Raposo — PUC-Rio

Homologada em 08/03/07, conforme Ata No 005/2007 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 16 – sala 106 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

Dados Internacionais de Catalogação na Publicação (CIP)

R549m Rieder, Rafael

Uma metodologia par especificar interação 3D utilizando Redes de Petri / Rafael Rieder. – Porto Alegre, 2007.
98 f.

Diss. (Mestrado em Ciência da Computação) – Fac. de Informática, PUCRS.
Orientação: Prof. Dr. Márcio Serolli Pinho.

1. Informática. 2. Redes de Petri. 3. Computação Gráfica Tridimensional. 4. Orientação a Objetos. I. Pinho, Márcio Serolli.

CDD 005.1

Ficha Catalográfica elaborada pelo Setor de Processamento Técnico da BC-PUCRS

PUCRS
Campus Central
Av. Ipiranga, 6681 – prédio 16 – CEP 90619-900
Porto Alegre – RS – Brasil
Fone: +55 (51) 3320-3544 – Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

Aos meus pais.

Agradecimentos

Primeiramente, gostaria de agradecer a Deus por iluminar meu caminho durante a jornada da vida. Com fé e determinação, foi possível chegar a mais um porto seguro.

Um agradecimento todo especial a meus pais, Adelar e Vera, pelo apoio e incentivo constante, tanto nas horas de bonança, como nos momentos difíceis. Pais, saibam que vocês são as pessoas mais importantes de minha vida e sou eternamente grato por tudo! Um carinho especial também ao Rômulo e ao Rolf que, além de meus irmãos, são meus melhores amigos.

Agradeço ao meu orientador Dr. Márcio Serolli Pinho pela experiência em ser seu aluno de Mestrado. Além de ser um excelente e competente profissional, é um grande amigo, incentivador e exemplo para minha caminhada acadêmica. Um grande abraço, estendido aos demais professores do PPGCC-PUCRS.

Agradeço aos colegas do GRV, pelo aprendizado, amizade e alegria. Primeiramente, ao grande amigo Mauro, com quem aprendi muito sobre programação "otimizada", e pelas longas conversas e trabalhos realizados durante o Mestrado. A dupla dinâmica, Felipe e André, por terem me apresentado peripécias da Computação Gráfica e Realidade Virtual. Ao Régis e ao trio RPGEdu, João, Genilson e Luciane, um muito obrigado. E um "oooi" ao Pedro, pela sua extrema filosofia de vida.

Agradeço aos amigos com que dividi apartamentos, obtendo uma baita experiência de vida, tanto em Porto Alegre, como no Rio de Janeiro. Em especial, um muitíssimo obrigado ao Bogoni (Leandro), ao Codorna (Alexandre), a Mana (Mariana) e ao Leo (Leonardo).

Agradeço também aos amigos Márcio, Hugo, Gustavo e Cristian pelos papos, futebol, mates, cervejadas e churrascadas. Em nome destes, um efusivo abraço aos demais mestrandos e doutorandos. Um abraço também ao pessoal do Tecgraf e do CDPe. Valeu galera!!!

Um agradecimento especial aos meus amigos do peito, Digo (Rodrigo) e Guto (Gustavo), pela força, alento e confiança depositados. Valeu Knigthz, brows de sangue!!! Aos meus amigos Silvano e Leândro, um forte abraço. A Leandra, Madalena e Laerte, um carinho e agradecimento do fundo do coração por toda a vivência e convivência.

Ao povo peruano, um carinho sem par. Em nome dos meus amigos Carlos Enrique Portugal Poma e Jorge Isaac Rodriguez-Rodriguez, um abraço sem fronteiras desejando sucesso a todos que aqui estão completando seus estudos. Ah, amigos, estou com saudades do basquete!

Agradeço a CAPES, a PUC-Rio/PETROBRAS e ao Convênio DELL/PUCRS pelo apoio financeiro.

Finalmente, agradeço aos ensinamentos do Escotismo pelo Acauã, ao tradicionalismo Gaúcho, ao Grêmio FBPA e ao bom e velho Rock'n'Roll!!!

Resumo

Este trabalho apresenta uma metodologia para modelar e construir tarefas de interação 3D em ambientes virtuais usando Redes de Petri, uma taxonomia de decomposição de técnicas e conceitos de orientação a objetos. Para tanto, um conjunto de classes e uma biblioteca gráfica são requisitos para construção de uma aplicação e para controlar o fluxo de dados da rede. Operações podem ser desenvolvidas e representadas como nodos de uma Rede de Petri. Estes nodos, quando interligados, representam as etapas do processo interativo. A integração destas abordagens resulta em uma aplicação modular, baseada no formalismo de Redes de Petri que permite especificar uma tarefa de interação, e também o reuso dos componentes existentes em novos projetos de ambientes virtuais.

Palavras-chave: tarefas de interação, Redes de Petri, especificação.

Abstract

This work presents a methodology to model and to build 3D interaction tasks in virtual environments using Petri nets, a technique-decomposition taxonomy and object-oriented concepts. Therefore, a set of classes and a graphics library are required to build an application and to control the net dataflow. Operations can be developed and represented as Petri Net nodes. These nodes, when linked, represent the interaction process stages. The integration of these approaches results in a modular application, based in the Petri Nets formalism that allows specifying an interaction task, and also to reuse developed blocks in new virtual environments projects.

Keywords: interaction tasks, Petri nets, specification.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

RV	Realidade Virtual	23
RdP	Redes de Petri	23
MEF	Máquina de Estado Finito	23
AV	Ambiente Virtual	23
TI	Técnica de Interação	23
RA	Realidade Aumentada	35
XML	eXtensible Markup Language	36
RdP-C	Redes de Petri Coloridas	43
RdP-H	Redes de Petri Hierárquicas	43
OpenGL	Open Graphics Library	55
GLUT	OpenGL Utility Toolkit	55

Sumário

1 Introdução

O desenvolvimento de aplicações de Realidade Virtual (RV), em especial aquelas destinadas à pesquisa científica, ainda utilizam processos de modelagem e implementação pouco estruturados e formais, levando, na maioria dos casos, a reescrita de código e a dificuldade em obter uma análise da aplicação antes de seu desenvolvimento.

Para o entendimento de uma aplicação computacional, é útil usar alguma ferramenta de descrição formal que defina comportamentos, tais como Redes de Petri (RdP), *Unified Modeling Language* (UML) e Máquinas de Estado Finitos (MEFs). Estas ferramentas permitem compreender e avaliar cada etapa de funcionamento do sistema, além de possibilitar a geração automática de código a partir de diagramas e a modelagem em diferentes níveis de abstração.

Smith [43] destaca que a ausência de uma descrição formal em Ambientes Virtuais (AVs) dificulta a avaliação de semelhanças entre Técnicas de Interação (TIs) diferentes, o que acaba levando a "reinvenção" de técnicas. Além disso, conforme aborda Navarre [24], descrições informais facilitam ambigüidades nas implementações.

Formalismos já têm sido usados para modelar TIs e tarefas de interação [24]. Hynet [49], ICO [30] e Flownet [50] são alguns exemplos de formalismos existentes, baseados em RdP. O uso destes formalismos ajuda, por exemplo, na detecção de falhas em sistemas ainda em tempo de projeto.

Além de formalismos, pesquisadores procuram desenvolver taxonomias capazes de documentar e especificar AVs num nível de detalhe mais próximo da concepção do usuário. Conforme Lindeman [19], a RV ainda é uma tecnologia que está em fase de definição e necessita ser classificada e categorizada. De acordo com Bowman [7], esta classificação e categorização permitiria entender o conjunto de TIs utilizadas no desenvolvimento de AVs. Uma vez identificadas, poderiam também ser usadas na organização do projeto interativo.

Os trabalhos que apresentam taxonomias procuram identificar as etapas do processo interativo [7], classificar TIs [8] [19] [38] e organizar o controle de sistema [9]. Estas abordagens provêm a separação de contextos, dividindo o sistema em partes menores (componentes), responsáveis por encapsular uma determinada funcionalidade. Isto permite que componentes sejam reutilizados por novos projetos, possibilitando a combinação destes com o objetivo de criar novas TIs, por exemplo.

Tanto o emprego de formalismos, como de taxonomias, visam otimizar o tempo de projeto e desenvolvimento de AVs. Uma proposta de **integração** destas abordagens pode permitir a especificação de sistemas de acordo com o nível de conhecimento do usuário, além de possibilitar o detalhamento de cada etapa do processo de desenvolvimento de *software*.

Baseado nestes trabalhos anteriores, esta dissertação descreve uma metodologia capaz de modelar e implementar componentes que representem as etapas do processo interativo. A união destes componentes permite a descrição de uma tarefa de interação, onde a seqüência de passos a serem desenvolvidas pelo usuário é controlada por uma RdP.

Esta metodologia integra três abordagens de modelagem: o formalismo de RdP, a taxonomia de decomposição proposta por Bowman [7] e conceitos de orientação a objeto. RdP são utilizadas para representar graficamente o comportamento de um AV com base na divisão das tarefas do processo interativo da taxonomia de decomposição de Bowman. O uso destas abordagens gera um modelo que pode ser codificado com auxílio de um conjunto de classes em C++, o qual define a estrutura e o funcionamento do sistema.

A escolha de RdP para especificação de tarefas em AVs surge naturalmente quando se utiliza uma taxonomia como a de Bowman, pois as tarefas de interação podem ser interpretadas como transições, enquanto os estados assumidos pela aplicação podem ser modelados como lugares na RdP. A partir dessa interpretação, é possível definir componentes independentes capazes de representar determinadas funcionalidades.

Um artigo com a descrição e aplicação da metodologia de especificação desenvolvida foi submetido para a conferência *IEEE Symposium on 3D User Interfaces 2007*. Este artigo pode ser lido no Apêndice A.

O presente texto está organizado da seguinte forma: O Capítulo 2 apresenta uma análise de trabalhos relacionados de métodos e técnicas de especificação existentes na literatura, detendo-se naqueles voltados ao domínio de AVs e TIs. O Capítulo 3 descreve as abordagens utilizadas para concepção da metodologia, enquanto que o Capítulo 4 apresenta base da metodologia proposta. O Capítulo 5 aplica esta metodologia, demonstrando os passos necessários para a modelagem e a geração de código que implementa o AV, além de apresentar testes que validam a especificação. Finalmente, o Capítulo 6 apresenta as conclusões deste trabalho, relatando também os objetivos que se pretende alcançar com trabalhos futuros.

2 Trabalhos Relacionados

Diferentes mecanismos têm sido propostos pela comunidade de RV para descrever e implementar tarefas de interação, buscando compreender a dinâmica das aplicações e possibilitar a padronização de funcionalidades.

Este Capítulo apresenta algumas propostas de especificação do processo interativo para AVs. Metodologias e técnicas que serviram de base para a concepção deste trabalho são brevemente relatadas, destacando objetivos particulares, vantagens e desvantagens de cada uma delas.

2.1 HyNet

HyNet [49] (do inglês *Hybrid High-level Petri Nets*) é uma metodologia de especificação de interfaces e TIs que integra três abordagens de modelagem bem fundamentadas na literatura. RdP de alto nível representam a base formal da especificação, definindo a semântica e permitindo a representação gráfica dos eventos discretos da aplicação. Equações diferenciais, por sua vez, permitem a descrição do comportamento contínuo do sistema, enquanto os conceitos de orientação a objeto permitem aumentar o poder de expressividade da metodologia, proporcionando modelos sucintos e compactos.

Conforme Smith [42], HyNet é um formalismo voltado ao projeto de interfaces. Seus modelos visam auxiliar o projetista na identificação de problemas de interfaces e na escolha adequada de técnicas a serem empregadas, posteriormente, nas etapas de desenvolvimento de protótipos e da própria aplicação. O formalismo também permite a representação do modelo em diferentes níveis de hierarquia, facilitando sua compreensão. A notação de um modelo HyNet baseia-se em uma **Rede de Petri Objeto Relacional Hierárquica Temporizada** [49], que é composta por oito elementos básicos, conforme mostra a Figura 1. Nesta metodologia, os diferentes tipos de arcos determinam o fluxo de execução da rede que, conseqüentemente, define o que será apresentado ao usuário.

Nesta metodologia, os **lugares** definem as classes do sistema. Sua descrição determina quais são os tipos de objetos (**Marcas**) suportados. **Transições** definem características, e podem ser tanto discretas como contínuas. **Transições contínuas** dispõem de um conjunto de equações diferenciais para representar os diferentes comportamentos da aplicação. **Transições discretas** podem ter associado a elas o tempo de disparo e a forma de propagação das ações realizadas. **Arcos** ligam lugares e transições, determinando regras para execução da rede. Eles podem ser

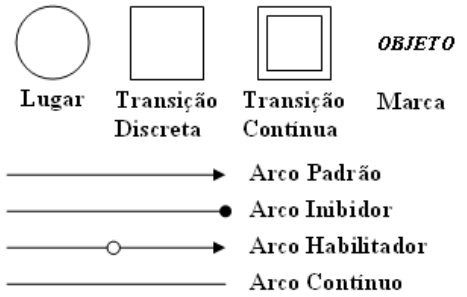


Figura 1 – Elementos de modelagem para o formalismo HyNet.

arcos padrões (indicam o fluxo de marcas, sendo pré e pós-condição de uma transição discreta), **arcos inibidores** (determinam marcas que bloqueiam uma transição), **arcos habilitadores** (determinam marcas que ativam uma transição) e **arcos contínuos** (mesmo que arcos padrões, só que para transições contínuas).

Para ilustrar o uso deste formalismo em ambientes virtuais (AVs), as Figuras 2 e 3 apresentam uma especificação da técnica de navegação baseada em *mouse*. Esta técnica realiza um simples vôo de câmera sobre um plano, com alterações na posição relativa do *mouse* e na velocidade do movimento. A Figura 2 é uma representação em alto nível da técnica, que procura abstrair detalhes das funcionalidades executadas, visando facilitar a compreensão do modelo (arcos pontilhados indicam que diferentes tipos de arcos estão sendo utilizados). *Mouse Movement* e *Scene* são definidos como os pontos iniciais da rede. Já a Figura 3 apresenta o detalhamento da operação de *mouse*, representada na Figura 2 pela transição *Mouse Movement*. Nesta Figura, as três abordagens do formalismo estão presentes: elementos de uma RdP, equações diferenciais (conteúdo das transições) e orientação a objetos (esboço da classe).

O formalismo HyNet facilita a representação do comportamento das TIs. Suas descrições auxiliam as fases de análise e projeto de AVs e podem servir de base para a implementação destas aplicações. No entanto, a proposta de especificação detalhada pode dificultar a representação de AVs, principalmente aqueles que envolvem diferentes técnicas e dispositivos (multimodais), levando a modelos complexos e "poluídos" visualmente, mesmo se for feito uma modelagem hierárquica. Além disso, o formalismo não proporciona uma forma de geração automática de código, o que requer o aprendizado da metodologia tanto por parte de projetistas como de desenvolvedores.

2.2 Flownet

Flownet [43] [50] é uma técnica de especificação formal desenvolvida para a modelagem de TIs, baseada na metodologia HyNet. O formalismo Flownet usa RdP para definir o comportamento discreto de TIs e elementos de uma notação para modelagem de sistemas dinâmicos para descrever o comportamento do fluxo de dados contínuo das técnicas e objetos do AV.

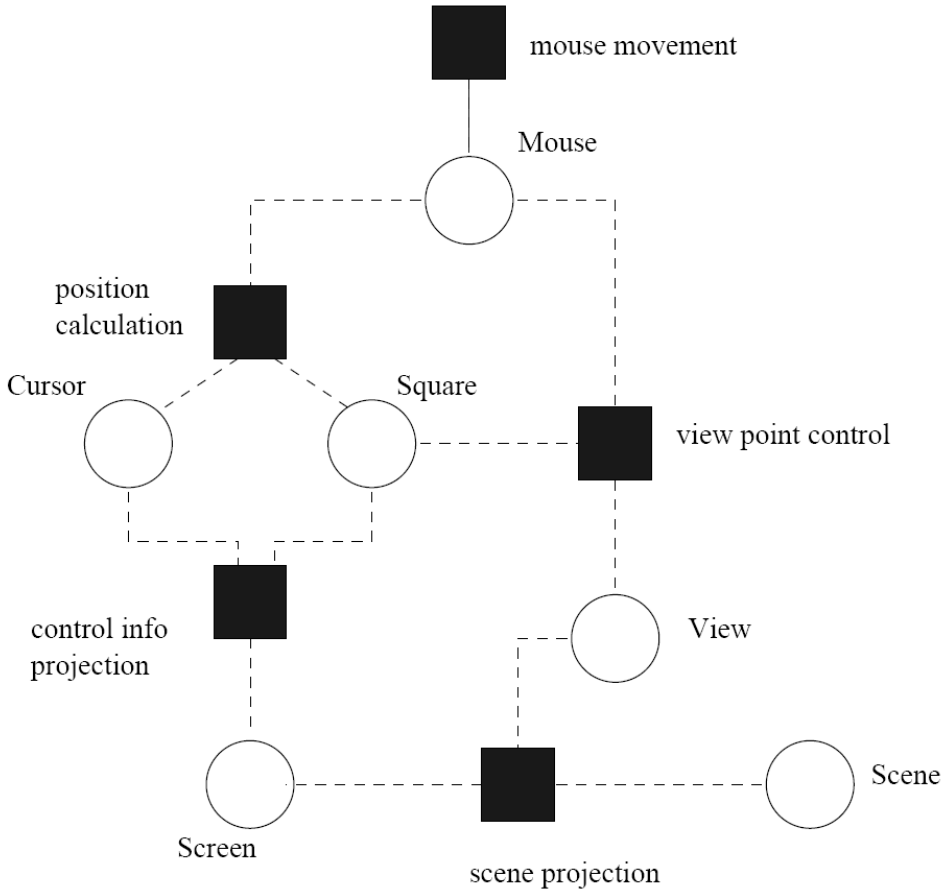


Figura 2 – Modelo HyNet hierárquico de uma interface de navegação (adaptado de Smith e Duke [43]).

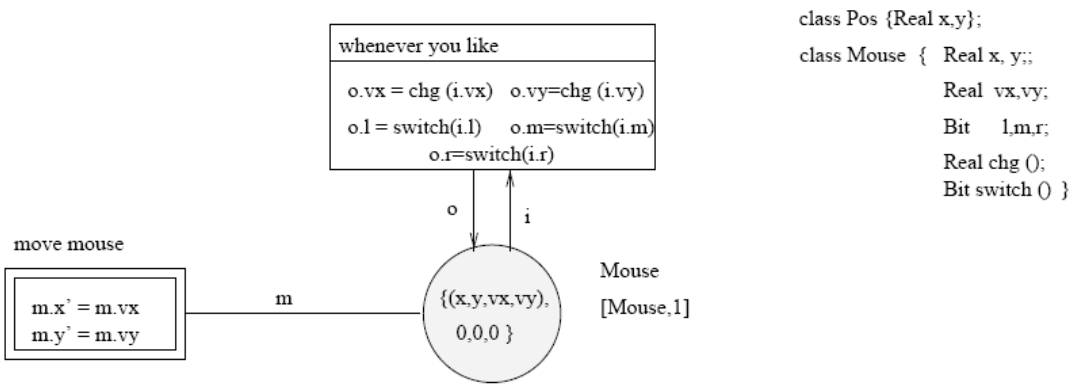
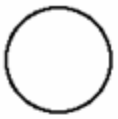
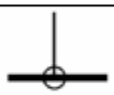


Figura 3 – Sub-rede representada pela transição *Mouse Movement* da Figura 2 (adaptado de Smith e Duke [43]).

Conforme Smith e Duke [43], o uso de Flownet tem por objetivo ajudar na descrição e compreensão dos componentes de um AV. Esta técnica de modelagem oferece uma notação de interface entre o sistema e o usuário que possibilita a representação dos fluxos de dados. Esta notação é composta pro dez componentes gráficos para especificação de sistemas. A Tabela 1 apresenta estes componentes, e brevemente relata a funcionalidade de cada um no modelo Flownet.

Tabela 1 – Componentes da notação Flownet (adaptado de Willams e Harrison [50]).

Nome	Símbolo	Descrição
Estado		Estado discreto usado em Redes de Petri. Pode estar ativo ou inativo. Pode servir com a representação de uma variável que armazena um *valor lógico* a ser usado em um programa. **Entradas:** Arcos vindos de transições. **Saídas:** Arcos para transições ou controles de fluxo
Transições		Transição discreta usada em Redes de Petri. A transição é disparada quando todos os arcos que nela chegam vêm de estados ativos. Se houver um arco inibidor chegando na transição, o estado a ele associado deve estar desativado. **Entradas:** arcos ou arcos inibidores. **Saídas:** arcos para estados
Arco de Controle		Assinala a dependência entre um componente e outro. Pode iniciar em Estados, Transições e Sensores de Fluxo. Pode terminar em Estados, Transições e Controladores de Fluxo
Arco Inibidor		Funciona como o Arco de Controle; entretanto, uma condição habilitada em seu início serve como um inibidor do elemento ligado ao seu final
Fluxo Contínuo		Representa o fluxo de informações que deve ser considerado contínuo
Controle de Fluxo		Habilita ou desabilita um Fluxo de Contínuo de dados, funcionando como uma válvula pela qual passa um Fluxo Contínuo. A *abertura* e o *fechamento* da válvula são definidos por Arcos de Controle ou Arcos Inibidores. **Entradas:** um Fluxo Contínuo e um ou mais Arcos de Controle e Inibidores. **Saídas:** Fluxo Contínuo
Sensor		Seleciona uma condição a partir de um fluxo contínuo. Esta condição gera um valor lógico a ser usado como entrada em uma transição ou em uma válvula de um Controle de Fluxo
Armazenagem		Fonte ou repositório de dados consumidos ou gerados em um Fluxo Contínuo. Pode servir como a representação de uma variável numérica que armazena um valor a ser usado em um programa. **Entradas:** um ou mais fluxos contínuos. **Saídas:** um ou mais fluxos contínuos
Transformador		Transformações a serem aplicadas aos dados de um fluxo contínuo. Podem representar rotinas a serem usadas em um programa. **Entradas:** um ou mais fluxos contínuos. **Saídas:** um ou mais fluxos contínuos
Conexões externas		Indica a fonte ou um destino de um Arco ou Fluxo Contínuo. Esta fonte é externa ao dado diagrama. Pode ser usado tanto para representar uma informação externa como os dados de um mouse ou rastreador de posição, quanto para servir de conexão com outros diagramas.

A especificação usando Flownet procura definir a seqüência dos possíveis comportamentos apresentados pelo sistema, sem preocupar-se com detalhes de implementação. Isto abstrai a complexidade do sistema, permitindo o refinamento das técnicas utilizadas e a redução do mo-

delo. Esta redução facilita a compreensão dos requisitos do sistema, e como estes podem ser alcançados durante a fase de projeto, resultando em modelos compactos, objetivos e expressivos.

O formalismo também dispõe de uma ferramenta, denominada Marigold [51], que possibilita ao desenvolvedor a construção dos modelos e protótipos. A ferramenta permite a conexão de componentes que definem TIs e dispositivos com um AV para simulação dos comportamentos. A Figura 4 apresenta um exemplo que modela uma técnica de navegação por vôo, usando Marigold, onde o usuário utiliza suas mãos para controlar seu deslocamento.

Como visto, Flownet oferece um método sistemático para projeto, teste e refinamento de TIs, com a possibilidade de prototipação de AVs. No entanto, mesmo oferecendo recursos mais intuitivos para representação de técnicas e dispositivos, e uma ferramenta de geração de código, Flownet continua por exigir o aprendizado da metodologia. De acordo com os trabalhos estudados, os protótipos gerados referem-se apenas à AVs simples, que envolvem um número reduzido de objetos, técnicas e dispositivos. Além disso, a ferramenta Marigold não está disponível para uso, seja comercial ou gratuito.

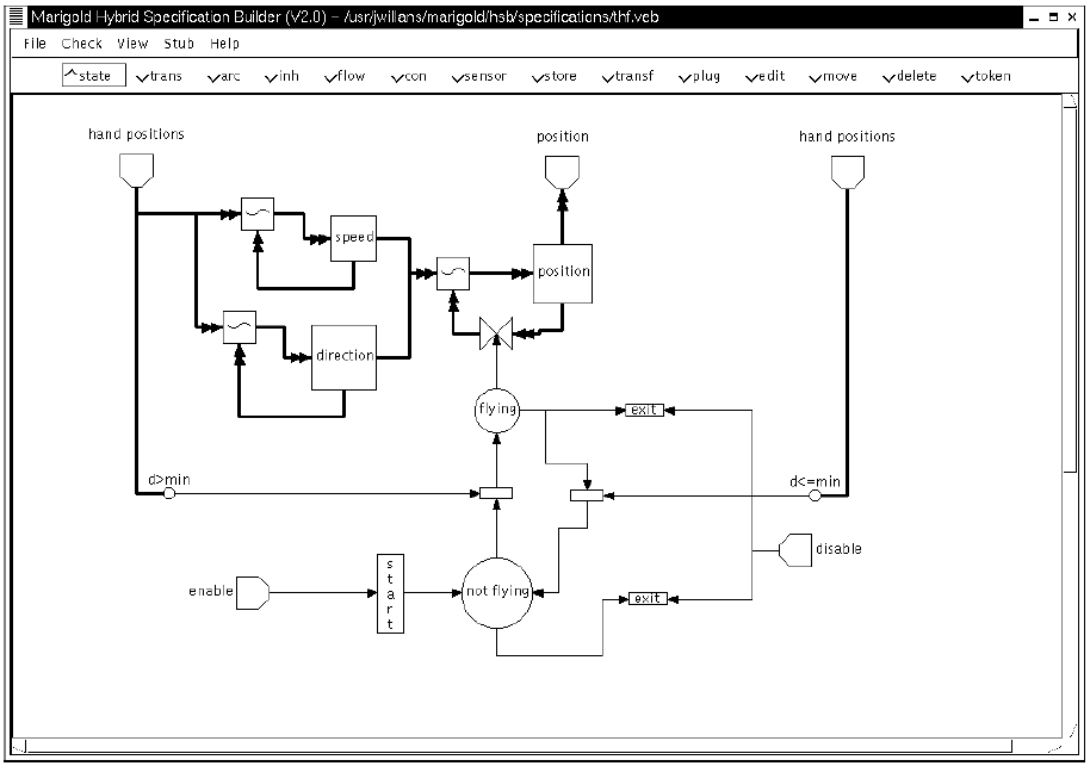


Figura 4 – Especificação Flownet da técnica de navegação por vôo usando duas mãos virtuais, na ferramenta Marigold (adaptado de Willams e Harrison [50].

2.3 ICO

O formalismo ICO (do inglês *Interactive Cooperative Objects*) é uma notação formal destinada à especificação de sistemas interativos [2], que usa conceitos de orientação a objetos para descrição dos aspectos estruturais ou estáticos de sistemas e RdP de alto nível para a descrição dos aspectos dinâmicos ou comportamentais.

Conforme Palanque [29], ICO foi planejado originalmente para modelagem e implementação de interfaces gráficas baseadas em eventos. O modelo ICO de um sistema é composto de objetos cooperativos [25], onde o comportamento dos objetos e o protocolo de comunicação entre eles são descritos por uma RdP.

Neste formalismo, um objeto é representado por uma entidade caracterizada por quatro componentes:

- **Comportamental:** define como o objeto deve reagir aos estímulos externos do sistema, através de uma RdP de alto nível. Em outras palavras, são as opções de interação oferecidas pelo sistema ao usuário;
- **Serviços:** define a interface de comunicação entre o objeto e o ambiente, e como estes estão interligados (funções de ativação);
- **Estado:** permite definir a disponibilidade dos serviços, baseado nas informações que recebe do usuário e do sistema;
- **Apresentação:** é a visão que o usuário tem do sistema e com a qual ele interage. É o componente responsável pelo *rendering* da aplicação.

ICO é utilizado para fornecer uma descrição formal do comportamento de uma aplicação interativa, mapeando as possíveis interações que um usuário pode ter acesso. Esta especificação compreende tanto a descrição das ações do usuário que influenciam a aplicação, como aquelas que são destinadas a apresentar ao usuário informações pertinentes sobre sua interação.

Sua especificação é executável, e oferece a possibilidade de prototipar e testar uma aplicação antes que ela seja completamente implementada [25]. O formalismo também considera separadamente a modelagem de TIs e de dispositivos usados. Além destas vantagens, a especificação permite que os modelos sejam validados com o uso de ferramentas de análise e de prova desenvolvidas pela comunidade de RdP que dão suporte a objetos cooperativos.

As Figuras 5 e 6 são exemplos de modelos ICO que representam AVs. A Figura 5 representa a ação de pegar e soltar uma peça de um jogo de xadrez usando uma luva de RV, enquanto que a Figura 6 representa a ação de controlar os instrumentos de uma simulação de vôo usando comandos de voz e gestos.

Apesar dos novos conceitos propostos por ICO suportarem a especificação de sistemas multimodais e a simulação dos modelos, sua abordagem também não oferece uma forma de geração de código, assim como nos formalismos HyNet e FlowNet. Conseqüentemente, projetistas

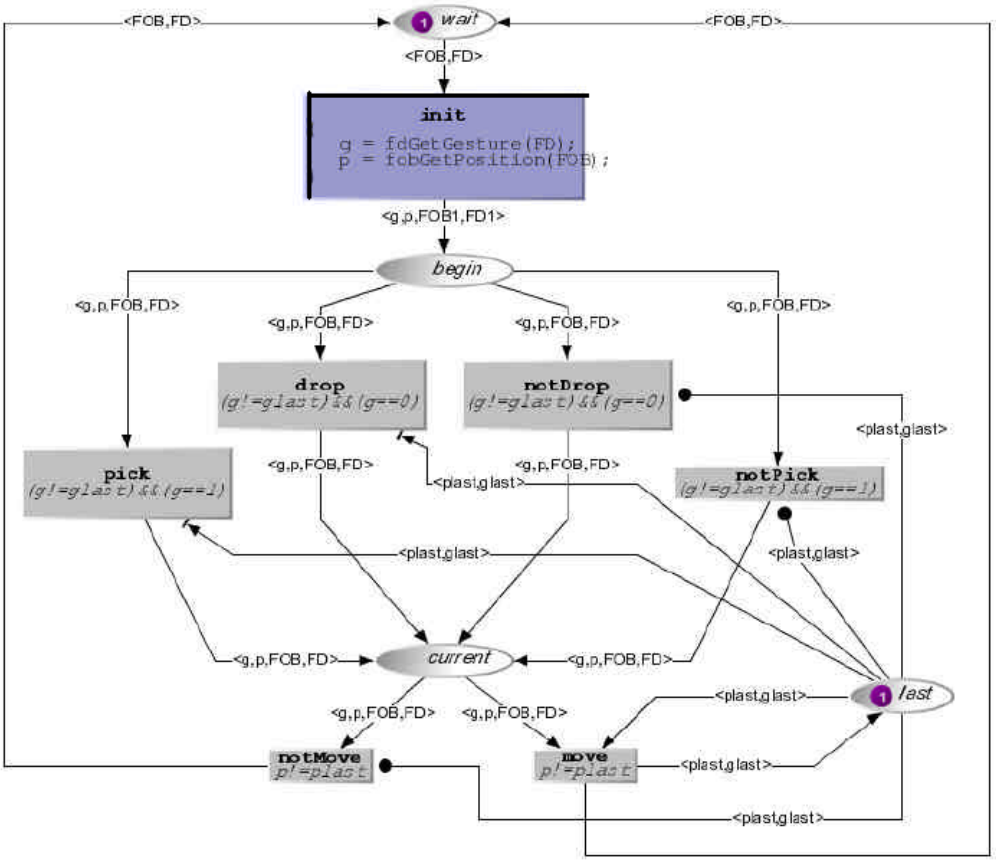


Figura 5 – Representação da captura de movimentos (pegar, mover e soltar) de um AV, usando uma luva de RV (adaptado de Navarre [24]).

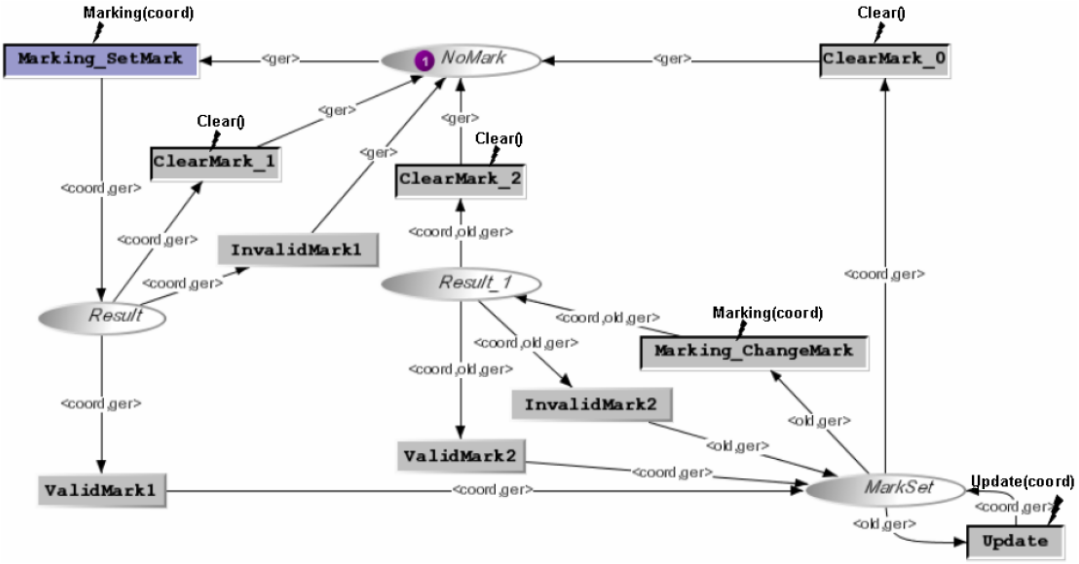


Figura 6 – Representação do controle de diálogo do piloto com as funcionalidades do AV que simula um vôo militar (adaptado de Bastide [1]).

e desenvolvedores necessitam conhecer todos os recursos do formalismo para concepção dos sistemas de RV.

2.4 InTML

Figueroa [11] propõe uma arquitetura de desenvolvimento de TIs, baseada em filtros, no qual fontes de informação (como, por exemplo, dispositivos) geram um fluxo de dados que são propagados entre filtros interconectados.

Neste trabalho, a linguagem de marcação InTML (do inglês *Interaction Techniques Markup Language*), baseada no X3D [53], foi desenvolvida para servir de *front-end* às bibliotecas de RV. Esta linguagem foca a integração do comportamento específico da aplicação com o comportamento dos objetos e eventos de dispositivos de entrada, permitindo que TIs sejam construídas e tratadas como componentes externos, independentes da aplicação. Isto permite que técnicas possam ser integradas, criando novas formas de interação.

Nesta arquitetura, um filtro representa qualquer dispositivo, técnica de interação, comportamento ou conteúdo de uma aplicação de RV descrito por um arquivo InTML. Sua interface é definida por portas de entrada e saída as quais são os tipos de eventos que podem receber ou produzir dados, respectivamente. A Figura 7 apresenta um exemplo de filtro que representa a técnica de seleção por toque, onde suas entradas (à direita) recebem um objeto 3D que representa a mão do usuário, a posição e orientação de um objeto, a cena de objetos disponíveis para seleção e eventos que informam sobre a inserção ou exclusão de objetos da cena.

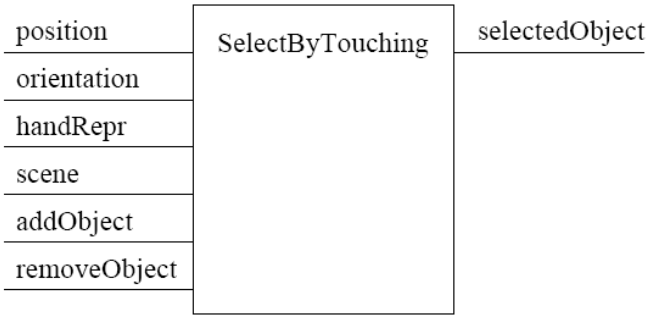


Figura 7 – Representação da técnica de seleção por toque como filtro InTML (adaptado de Figueroa [12]).

A execução de um filtro é dividida em três estágios principais:

- **Coleta de dados:** toda informação gerada em certo intervalo de tempo é coletada. Este estágio é considerado como de pré-processamento, no qual filtros selecionam e manipulam a informação que eles recebem, procurando preparar os dados para a próxima etapa;
- **Processamento:** neste estágio, um filtro executa de acordo com as informações de entrada e atualiza seu estado interno. Informação de saída é gerada, mas não propagada;

- **Propagação da saída:** a informação gerada pela etapa de processamento é transmitida a todos os filtros interconectados.

Filtros também podem ser combinados, gerando componentes que representam uma nova TI. A Figura 8 apresenta um exemplo que mostra a técnica Go-Go [37] sob o ponto de vista geral (à esquerda) e detalhado (à direita). Este recurso também permite que tais componentes sejam recombinados para gerar uma nova TI.

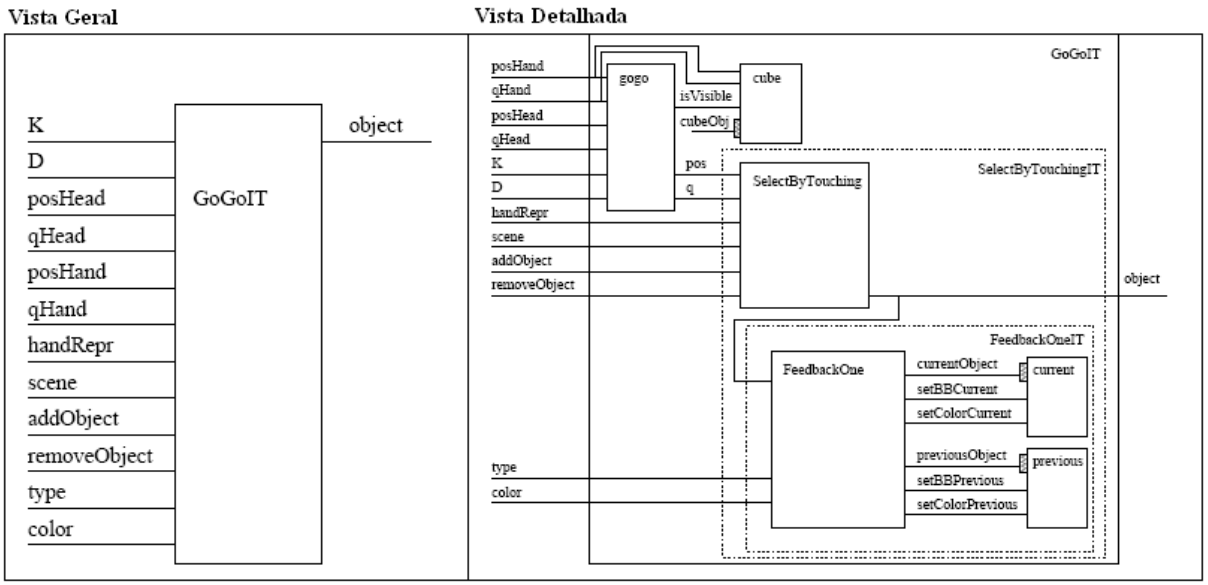


Figura 8 – Técnica de interação Go-Go, com vista geral e em detalhes (adaptado de Figueroa [12]).

Figueroa [11] destaca que, usando uma linguagem padronizada como InTML, projetistas de aplicações de RV podem entender e comparar técnicas numa mesma linguagem, sob um paradigma de documentação uniforme. A linguagem também permite abstrair a complexidade de um AV, e facilita o reuso do código de cada componente criado. No entanto, como o processo de geração de código, oferecido pela arquitetura, resulta em um código interpretado (no caso, Java), a qualidade da interação pode ficar comprometida caso não exista um *hardware* que suporte as exigências da aplicação.

2.5 Unit

O *framework* Unit [27] usa uma linguagem de programação de fluxo de dados para descrever TIs. Unit estabelece uma camada de abstração entre os dispositivos de entrada e a aplicação para TIs, permitindo ao desenvolvedor separar as funcionalidades da aplicação das técnicas e comportamentos do projeto de interfaces gráficas. Este *framework* foi desenvolvido usando a linguagem de programação Java, juntamente com a biblioteca Java3D [46], característica esta que o torna independente de plataforma.

Para criação da camada de técnicas, Unit usa um conceito de unidades que representam fluxos de dados. Estas unidades são componentes que, agrupados, procuram representar uma determinada TI. Recursos da aplicação e de dispositivos comunicam-se com esta camada, oferecendo subsídios para o funcionamento da TI.

Para visualizar as ligações entre as camadas da aplicação, Unit dispõe de uma interface gráfica que especifica a interação através de uma representação por grafos. A Figura 9 apresenta um exemplo de grafo que representa parte do fluxo de dados da TI *Flexible Pointer*.

O *framework* também permite que unidades que compõem uma TI estejam distribuídas em diferentes máquinas, através de comunicação remota. Esta característica possibilita, por exemplo, testes com dispositivos que estão permanentemente conectados a uma determinada estação de trabalho. Unit também possibilita a troca de técnicas durante a execução da aplicação.

No entanto, o uso de grafos para representar a interação pode dificultar a compreensão de TIs, a medida que a sua complexidade aumenta. Além disso, o *framework* também necessita um *hardware* robusto para a execução dos testes, uma vez que o código é interpretado.

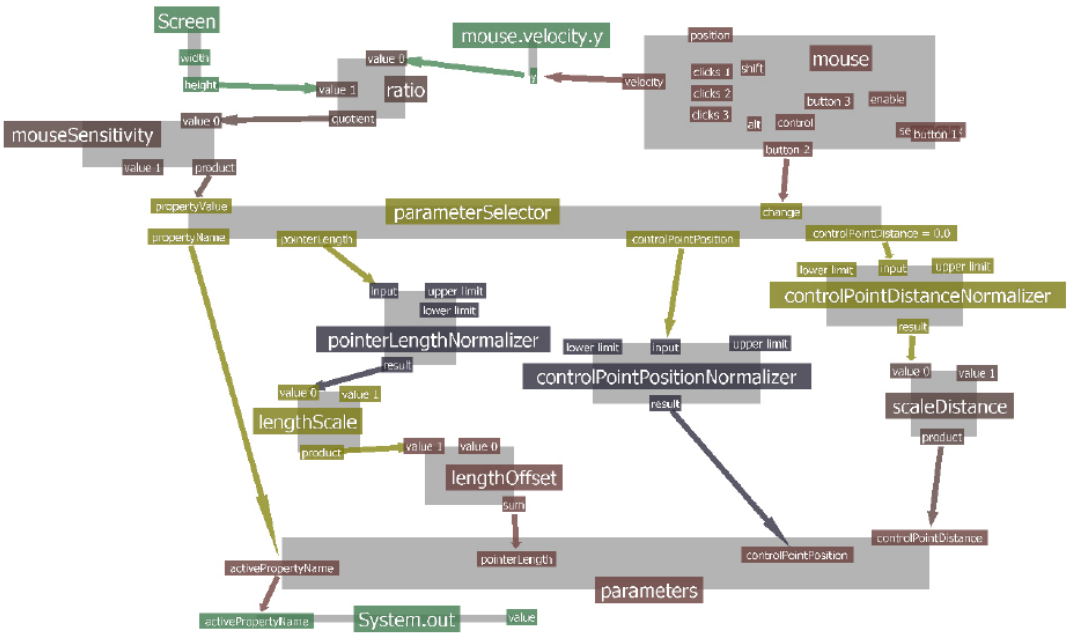


Figura 9 – Fluxo de dados que representa a TI *Flexible Pointer*, usando *mouse* e *joystick* (adaptado de Olwal e Feiner [27]).

2.6 CHASM

CHASM (do inglês *Connected Hierarchical Architecture of State Machines*) [52] é uma metodologia baseada em máquinas de estado hierárquicas que permite a comunicação entre o projetista e o programador de uma interface 3D, procurando descrever as características de sis-

temas interativos em forma de “conceitos reusáveis” que auxiliam no gerenciamento de código.

Conforme Wingrave [52], estes “conceitos reusáveis” são descrições sucintas do funcionamento de uma TI, que podem ser refinadas durante a fase de desenvolvimento, de acordo com as necessidades do projeto do AV. Conceitos estabelecidos podem ser armazenados em uma biblioteca, para posterior uso do desenvolvedor.

De acordo com o autor, esta forma de especificação permite que desenvolvedores expressem padrões de interação complexos, oferecendo uma forma de combinar TIs existentes com o mínimo esforço e sem alterar a base de uma aplicação.

O método de decomposição de técnicas em conceitos usado por CHASM adota o padrão “quando <condição> então <resposta>” para identificar um estado de comportamento. Os pesquisadores apresentam um exemplo deste método com o mapeamento do comportamento de uma aplicação que oferece duas TIs para o usuário: *World-In-Miniature* (WIM) [44] e *Ray-Casting*. Nesta configuração, para o uso de WIM o usuário deve posicionar sua mão em frente ao rosto. Em caso contrário, deve afastá-la. Em ambos os casos, a interação com um botão é necessária.

Uma implementação tradicional deste comportamento exigiria verificar o relacionamento entre a mão e a cabeça quando algum movimento de uma desta fosse detectado. Para isto, as seções de código precisam estar certas de que, no momento em que o botão for liberado, por exemplo, apenas uma das técnicas esteja habilitada. CHASM permite reconhecer estas situações em tempo de projeto, forçando o desenvolvedor a prever sua ocorrência durante a implementação e não em tempo de depuração.

CHASM também possibilita a geração de diagramas, como diagramas de transição e relacionamento, oferecendo recursos para a descoberta de erros, documentação e visualização do projeto do AV (veja Figura 10). No entanto, ainda não existem ferramentas capazes de manipular visualmente a base de conceitos, o que permitiria a concepção de novos conceitos e técnicas ainda na fase de projeto do AV. Além disso, o autor destaca a necessidade de métodos de avaliação cognitiva e o suporte à descrição de conceitos para AVs complexos, como AVs de Realidade Aumentada (RA).

2.7 IPN - IML

O trabalho de Ying [55] busca entender o processo interativo através da análise do código-fonte de aplicações de RV já implementadas. Usando os conceitos de engenharia reversa, são extraídas informações relacionadas à interação do usuário em AVs colaborativos. Tais informações são organizadas e armazenadas em um arquivo que serve de base para a geração de um modelo de RdP, denominado IPN (do inglês *Interaction Petri Net*).

O processo de extração analisa arquivos VRML/X3D, identificando eventos, relacionamentos e recursos por eles descritos. Uma lista de especificação armazena as informações coletadas

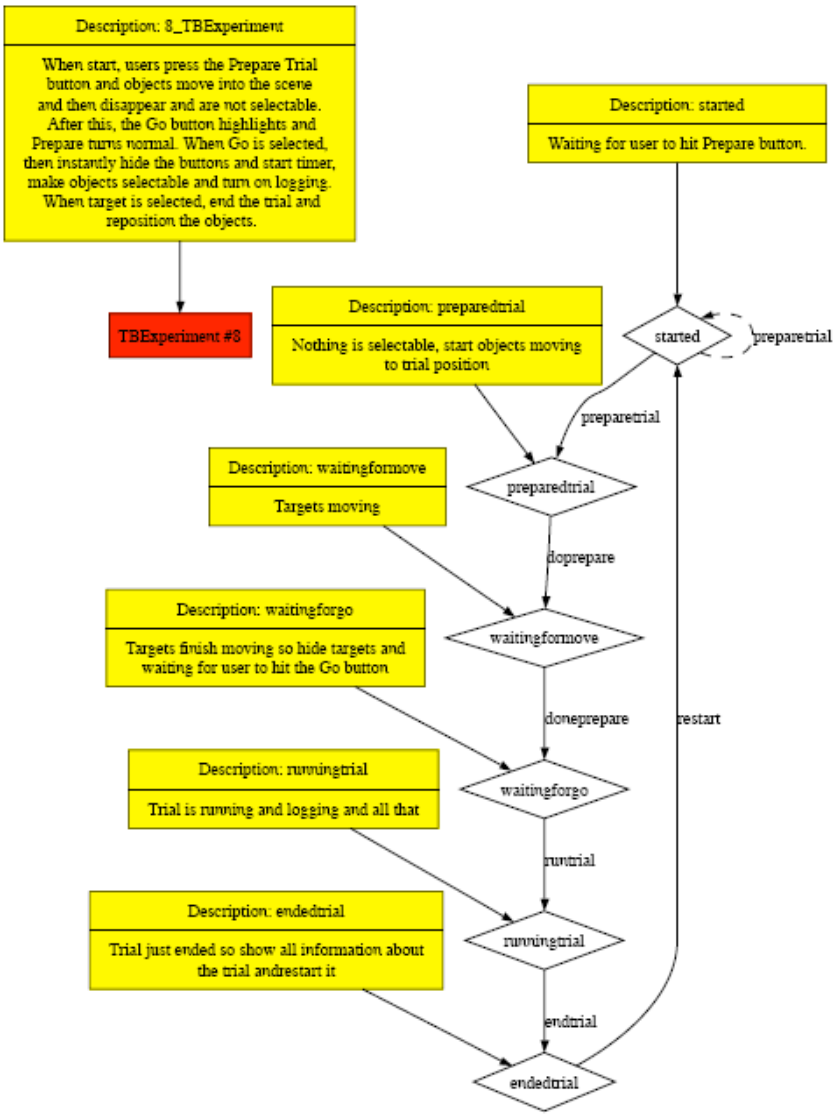


Figura 10 – Diagrama de transição gerado por CHASM (adaptado de Wingrave e Bowman [52]).

como objetos em uma linguagem de marcação baseada na XML, denominada IML (do inglês *Interaction Markup Language*). Estes objetos possuem diferentes estados para representar quando uma determinada interação do usuário está ocorrendo.

Para que uma ferramenta de modelagem possa compreender a especificação e gerar o modelo de RdP apropriado, uma tradução do arquivo IML é realizada para um novo arquivo, no formato PNML (do inglês *Petri Net Markup Language* [22]), usando para tal um vocabulário XSLT (do inglês *eXtensible Stylesheet Language Transformation*).

A Figura 11 ilustra o procedimento completo realizado por esta abordagem, onde (a) mostra o arquivo 3D analisado, (b) apresenta a lista de especificação e (c) exibe a RdP gerada.

Apesar da idéia de IPNs ser interessante para análise do processo interativo, a abordagem proposta pelos autores limita-se a descrição de AVs não-imersivos. Além disso, aplicações desenvolvidas em linguagem C/C++, freqüentemente utilizada na construção de AVs, não podem ser especificadas, restringindo ainda mais o domínio de uso da proposta.

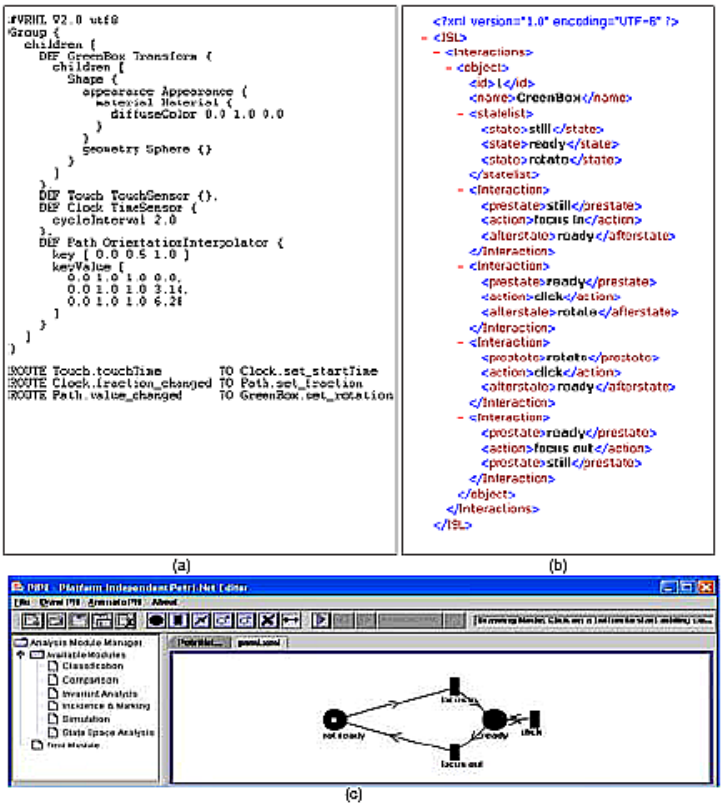


Figura 11 – Arquivo VRML analisado (a), arquivo de especificação IML (b) e modelo IPN gerado (c).

2.8 DWARF - *User Interface Control*

MacWilliams [21] apresenta uma arquitetura para interfaces de RA onde uma abordagem baseada em RdP permite a representação gráfica das metáforas de interação utilizadas por um sistema. Esta arquitetura tem por base o *framework* DWARF [4] (do inglês *Distributed Wearable Augmented Reality Framework*), responsável pelo controle de serviços de dispositivos de entrada/saída e protocolos de comunicação.

A interface de controle da aplicação comunica-se com o modelo (que representa a aplicação) através de funcionalidades oferecidas pela ferramenta de modelagem JFern [26]. Componentes de entrada enviam dados para a interface de controle, que os encapsula e envia para a rede. Esta efetua a simulação de acordo com as regras estabelecidas pelo modelo, e devolve para a aplicação a ação a executar.

Para demonstrar a utilização dos recursos desta arquitetura, os pesquisadores projetaram o sistema SHEEP [21], um jogo onde o objetivo é criar ovelhas e reuni-las em rebanhos através de uma interface de RA.

A Figura 12 apresenta esta aplicação, onde o usuário interage através de um bastão e comandos de voz, e suas ações são simuladas por uma RdP. No exemplo da Figura, enquanto não existe colisão no cenário (a), a RdP permanece ociosa, sem marcas depositadas em seus lugares. Quando um bastão toca a interface (b), uma marca é depositada no lugar L4. Quando é dado um

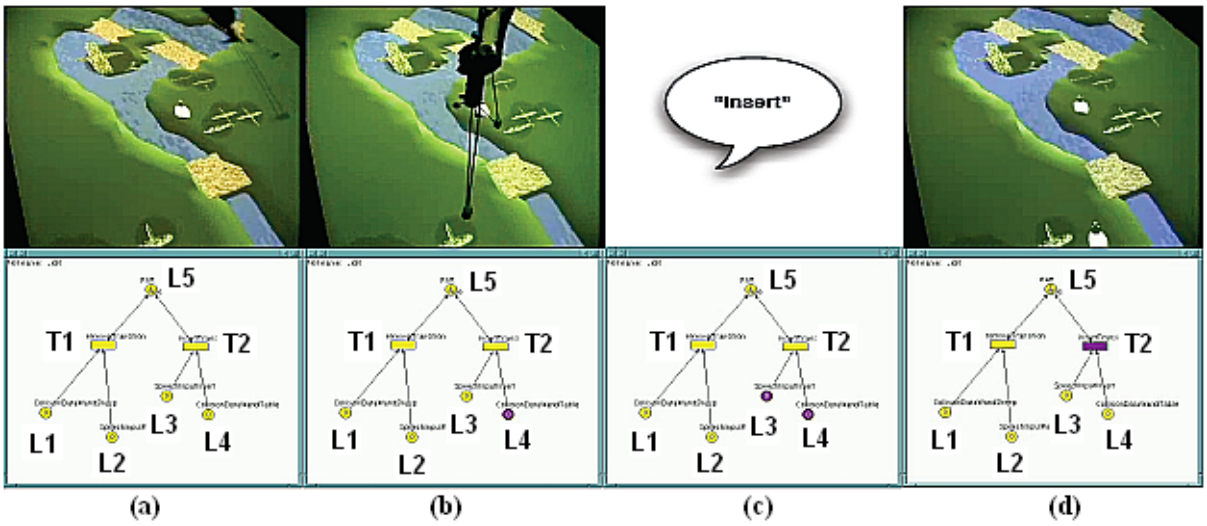


Figura 12 – Interface de RA do jogo SHEEP, onde um modelo de RdP representa as interações do usuário no AV.

comando de voz (“Insert”), uma marca é depositada no lugar L3. Existindo estas duas marcas, a transição T2 responsável por inserir uma nova ovelha no AV é disparada (d).

De acordo com Hilliges [15], representações gráficas do funcionamento da aplicação são úteis para o processo de *debugging*. Neste exemplo, o avaliador pode ter uma visão do processo interativo em execução na rede, enquanto o usuário interage normalmente com a aplicação.

As RdP que modelam as interações do usuário são descritas por arquivos XML, onde estruturas de classes podem ser geradas na linguagem Java. A ferramenta JFern [26] é evocada durante a execução da aplicação, simulando graficamente o comportamento da rede. No entanto, tais modelos podem ser tornar complexos e de difícil compreensão, a medida que o número de possibilidades de interação aumentam.

Procurando sintetizar o que foi apresentado neste Capítulo, a Tabela 2 apresenta um resumo das características e das fases de desenvolvimento abordadas em cada uma das técnicas.

Tabela 2 – Características das metodologias estudadas. As abreviações A, P e I da linha “Fases de Desenvolvimento” referem-se às etapas de Análise (A), Projeto (P) e Implementação (I) de sistemas.

	Características	**Hynet**	**Flownet**	**ICO**	**InTML**	**Unit**	**CHASM**	**IPN-IML**	**DWARF UIC**
1	Formalismo	RdP	RdP	RdP	—	—	MEF	RdP	RdP
2	Editor de Modelos	Não	Sim	Sim	Não	Não	Não	Não	Sim
3	Modelos Animados	Não	Não	Sim	Não	Não	Não	Não	Sim
4	Codificação	—	—	—	InTML	Grafos	Texto	VRML	XML
5	Geração de Código	—	—	—	Java	Java	Texto	XML	Java
6	Hierarquização	Sim	Sim	Não	Sim	Sim	Não	Não	Não
7	Fases do Desenvolvimento	A, P	A, P	A, P	A, P, I	A, P, I	A, P	I	A, P, I

3 Fundamentos da Metodologia

Pela análise dos trabalhos citados anteriormente, nota-se que as abordagens apresentam vantagens e objetivos particulares, mas, em geral, não atendem a contento todas as fase do ciclo de desenvolvimento de aplicações de computador. Neste sentido, este trabalho propõe uma metodologia para o desenvolvimento hierárquico de aplicações de RV, do projeto à implementação, com base no processo interativo.

Neste Capítulo são discutidas duas das três ferramentas utilizadas como fundamento para esta metodologia. Inicialmente, uma breve apresentação do **Formalismo de Redes de Petri** [32] aborda as principais definições, características, propriedades e extensões de modelagem presentes na literatura. Em seguida, é descrita a **Taxonomia de Decomposição de Tarefas de Interação 3D** proposta por Bowman [7]. Aspectos referentes à separação de tarefas de seleção, manipulação e navegação em partes menores são tratados por esta Seção.

Os conceitos de **Orientação a Objetos**, presentes nesta metodologia de especificação, não serão aqui abordados. Este paradigma de análise, projeto e programação de sistemas é amplamente e corriqueiramente utilizado por projetistas e desenvolvedores, além de estar bem fundamento pela literatura [41] [16] [5] [14] [13] [45] [10].

3.1 Redes de Petri

De acordo com Murata [23], uma Rede de Petri é uma ferramenta de modelagem gráfica e matemática aplicável a muitos sistemas, principalmente na descrição e estudo de sistemas concorrentes, assíncronos, distribuídos, paralelos, não-determinísticos ou estocásticos. Protocolos de rede, sistemas *fuzzy* e aplicações de RV são exemplos de sistemas que podem ser modelados e simulados utilizando esta ferramenta.

O nome "Redes de Petri" é uma homenagem a Carl Adam Petri, criador deste modelo em 1962. Em sua tese, em alemão, intitulada *Kommunikation mit Automaten* (em inglês *Communication with Automata*) [32], seu objetivo era desenvolver um modelo em que máquinas de estado fossem capazes de se comunicar.

Formalmente, uma RdP pode ser definida como uma quíntupla (P, T, A, w, M_0), onde:

- $P = \{p_1, p_2, ..., p_m\}$ é um conjunto finito de lugares;
- $T = \{t_1, t_2, ..., t_n\}$ é um conjunto finito de transições;

- $F \subseteq (P \times T) \cup (T \times P)$ é um conjunto de arcos;
- $w : F \rightarrow \{1, 2, ...\}$ é uma função que dá valor aos arcos;
- $M_0 : P \rightarrow \{0, 1, 2, ...\}$ é a marcação inicial da rede, com $(P \cap T) = \oslash$ e $(P \cup T) \neq \oslash$.

Raposo [39] destaca que, em um modelo de RdP, os estados estão associados aos lugares e suas marcações, e os eventos às transições. O comportamento de um sistema modelado por RdP é descrito em termos de seus estados e suas mudanças [23].

Graficamente, uma RdP é um tipo particular de grafo direcionado (Figura 13), composta por quatro tipos de elementos [31]:

- **Lugares ou *Places*:** representados por elipses, são vértices que representam os estados do sistema. São componentes passivos da RdP;
- **Fichas, Marcas ou *Tokens*:** simbolizados por pontos, representam a situação atual, ou seja, em que estado encontra-se o sistema modelado. As marcas iniciais informam o estado primitivo do sistema;
- **Transições:** vértices que representam às ações (eventos) do sistema, simbolizados por traços ou barras (horizontais ou verticais). Modelam o comportamento dinâmico do sistema;
- **Arcos:** interligam lugares a transições, e vice-versa, por meio de setas direcionais, indicando a seqüência de execução da rede. Transportam marcas entre os vértices do grafo, indicando as pré e pós-condições de uma transição. Também podem ter rótulos que definem o número de marcas a transportar de um vértice a outro.

A Figura 13 ilustra uma RdP que representa uma tarefa simples de um sistema de empacotamento [20]. O modelo descreve uma linha de montagem de um conjunto de porcas e parafusos, onde o número de peças disponíveis está representado por marcas nos lugares "Parafusos" e "Porcas", além de uma marca em "Máquina" indicando que a máquina de empacotar está livre para a execução da tarefa.

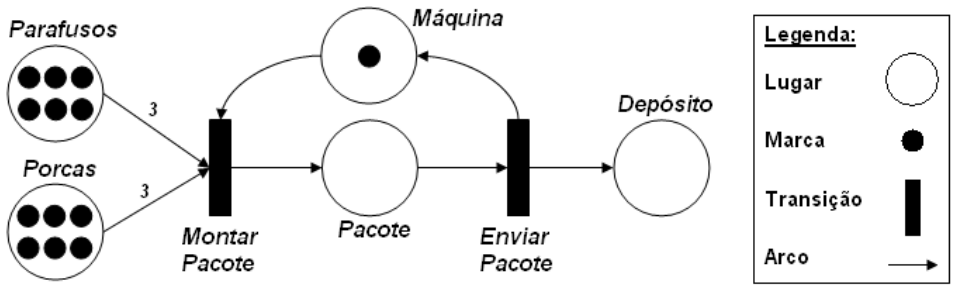


Figura 13 – Elementos de uma RdP, simulando uma linha de produção (adaptado de Maciel [20]).

A transição de “Montar Pacote” exige como pré-condição a existência de, no mínimo, três porcas e três parafusos (peso indicado nos arcos dirigidos), além da máquina liberada. Satisfazendo os requisitos, as marcas são consumidas, e uma nova marca será depositada no lugar “Pacote”. Esta etapa habilita a transição de “Enviar Pacote”, que recoloca uma marca em “Máquina” (indicando que a máquina está livre para novo empacotamento), enquanto “Depósito” recebe uma marca informando que mais um pacote está disponível ao setor de Expedição.

O exemplo abordado também poderia ser representado em notação matemática. Utilizando uma representação baseada em conjuntos, a Figura 13 poderia ser assim descrita:

- *P = {Parafusos, Porcas, Máquina, Pacote, Depósito}*;
- *T = {Montar Pacote, Enviar Pacote}*;
- *w(Parafusos, Montar Pacote) = w(Porcas, Montar Pacote) = 3*;
- *w(Máquina, Montar Pacote) = w(Montar Pacote, Pacote) = w(Pacote, Enviar Pacote) = w(Enviar Pacote, Máquina) = w(Enviar Pacote, Depósito) = 1*;
- M_0 *= [6 6 1 0 0].*

Maciel [20] também destaca que o emprego da álgebra matricial formaliza a teoria das RdP pois esta possibilita analisar o funcionamento da rede. A Figura 14 apresenta a descrição da Figura 13 usando este método matemático. Nesta Figura, A matriz **I** representa as pré-condições para cada uma das transições (colunas), e a matriz **O** define as pós-condições, atualizando os lugares da rede (linhas). Os números indicam o número de marcas que podem ser transportados de um determinado lugar para uma determinada transição.

$$
I = \begin{array}{|c|c|l|}
\hline
\text{Montar Pacote} & \text{Enviar Pacote} & \\
\hline
3 & 0 & \text{Parafusos} \\
3 & 0 & \text{Porcas} \\
1 & 0 & \text{Máquinas} \\
0 & 1 & \text{Pacote} \\
0 & 0 & \text{Depósito} \\
\hline
\end{array}
$$

$$
O = \begin{array}{|c|c|l|}
\hline
\text{Montar Pacote} & \text{Enviar Pacote} & \\
\hline
0 & 0 & \text{Parafusos} \\
0 & 0 & \text{Porcas} \\
0 & 1 & \text{Máquinas} \\
1 & 0 & \text{Pacote} \\
0 & 1 & \text{Depósito} \\
\hline
\end{array}
$$

Figura 14 – Matrizes de representação da RdP da Figura 13 (adaptado de Maciel [20]).

A representação matemática possibilita uma análise aprofundada do funcionamento do sistema, de acordo com determinadas propriedades identificadas pela teoria de RdP. Além disso, sistemas podem ser especificados com diferentes tipos de RdP existentes na literatura. As duas próximas subseções destacam, brevemente, estas outras características.

3.1.1 Propriedades

Conforme Murata [23] e Maciel [20], as propriedades das RdP podem ser divididas em dois grupos: propriedades estáticas (ou estruturais) e propriedades dinâmicas (ou comportamentais). O primeiro não depende das marcas e não se modifica durante o estágio de execução da rede. Já o segundo grupo depende das marcas, e tende a estar em constante atualização.

Abaixo, são apresentados os tipos de propriedades que compõem cada um desses grupos, e o que cada um deles procura evidenciar na rede.

De acordo com Murata [23], as propriedades comportamentais são:

- **Alcançabilidade (*reachability*):** indica a possibilidade de atingir-se um determinado lugar pelo disparo de um número finito de transições, a partir de uma marcação inicial. Satisfazendo esta propriedade, diz-se que a rede é alcançável, e todos os estados, em algum momento da execução do sistema, serão satisfeitos;

- **Limitação (*boundedness*):** define o número de marcas que cada lugar pode acumular. A passagem de um estado para outro depende do peso dos arcos associados às transições;

- **Segurança (*safeness*):** uma RdP é definida como segura se todos os lugares desta rede podem contar uma ou nenhuma marca. Esta propriedade está ligada à limitação da rede;

- **Vivacidade (*liveness*):** usada para constatar a ausência de *deadlocks*. Assim, esta propriedade é evidenciada quanto for possível executar todas as suas transições a partir de qualquer um dos estados alcançáveis da rede;

- **Cobertura (*coverability*):** identifica se uma transição é potencialmente disparável, isto é, se uma marca pode ser obtida a partir de uma outra, anterior a ela;

- **Persistência (*persistence*):** ocorre quando o disparo de uma transição não desabilita o disparo de outra. Este tipo de comportamento está presente em sistemas paralelos e circuitos digitais com atividades assíncronas;

- **Reversibilidade (*reversibility*):** ocorre quando o lugar inicial ou um grupo específico destes pode ser novamente alcançado;

- **Justiça (*fairness*):** uma RdP é dita justa se, para quaisquer duas transições, o número de vezes que uma é executada enquanto a outra não é executada é finito.

Já as propriedades estruturais são:

- **Limitação (*structural boundedness*):** uma rede é estruturalmente limitada se o número de marcas em cada lugar sempre permanece o mesmo;

- **Conservação (*conservation*):** ocorre quando a soma dos valores dos arcos que ligam as pré-condições de uma transição for igual à soma das pós-condições;
- **Repetitividade (*repetitiveness*):** essa propriedade se evidencia quando uma marcação habilita ilimitadamente uma seqüência de transições;
- **Consistência (*consistency*):** uma RdP apresenta essa propriedade quanto uma seqüência de transições é disparada, a partir da marcação inicial, retornando-a a esta configuração quando todas as transições da rede foram disparadas ao menos uma vez.

3.1.2 Extensões de RdP

O modelo original de RdP, abordado até esse momento, é simples e possibilita a representação de diferentes tipos de aplicação, como já destacado. No entanto, a medida com que pesquisadores adotaram RdP para representar seus sistemas, duas importantes características apresentaram-se de difícil modelagem: aspectos funcionais complexos (condições de fluxo de controle) e aspectos de temporização.

Para resolver esse problema, diversas extensões de RdP têm surgido na literatura [22]. Essas extensões, também chamadas de Redes de Petri de Alto Nível, procuram preservar a teoria inicial realizando adaptações conforme as necessidades e características do que se deseja modelar e formalizar. Dentre estas, duas freqüentemente são utilizadas, e servem de base para novas abordagens: *Redes de Petri Coloridas* (RdP-C) e *Redes de Petri Hierárquicas* (RdP-H).

RdP-C diferenciam-se do modelo original por permitir a distinção de marcas através de cores ou tipos de dados, permitindo que recursos sejam agrupados numa única representação, reduzindo conseqüentemente o tamanho do modelo.

De acordo com Jensen [17], uma RdP-C é uma representação gráfica e intuitiva que facilita a visualização da estrutura básica de um modelo complexo, além de permitir a compreensão individual do comportamento dos processos do sistema. Para tanto, elas são compostas por três diferentes partes:

- **Estrutura:** grafo direcionado com dois tipos de vértices (lugares e transições), com arcos valorados interconectando-os;
- **Declarações:** especificam o conjunto de cores e declarações de variáveis (legenda);
- **Inscrições:** identificam cada elemento da rede. Lugares têm nome, conjunto de cores e expressão de inicialização (marcação inicial). Transições podem possuir nomes e expressões associadas. Já inscrições em arcos podem ter tipos e expressões.

Conforme mostra a Figura 15, em uma RdP-C as marcas coloridas (ícones) podem especificar tipos de dados. Arcos podem definir os tipos suportados, ou são rotulados com condições

especiais, como, por exemplo, operações lógico-matemáticas sobre esses dados. Estas condições, chamadas de "guardas", também podem estar associadas a limitações impostas pelas transições, visando restringir o conjunto de dados a trabalhar.

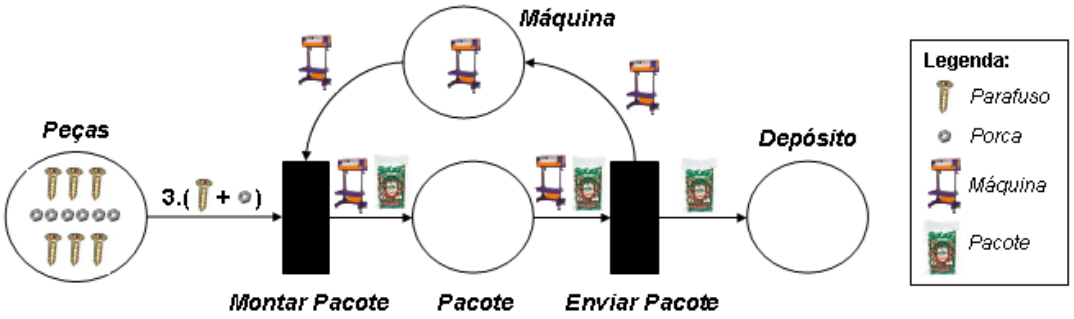


Figura 15 – Linha de produção simples representada por uma RdP-C.

Na Figura 15, os lugares "Parafusos" e "Porcas" estão modelados como um único lugar, "Peças", composto por ícones que simbolizam os recursos para a montagem dos pacotes. O arco entre "Peças" e "Montar Pacote" contém a inscrição que indica as condições mínimas para elaboração de um conjunto de peças, enquanto os demais arcos indicam a máquina ligada e o conjunto que foi produzido.

Já as RdP-H proporcionam uma descrição estruturada dos sistemas, possibilitando diferentes níveis de abstração que auxiliam na interpretação dos modelos. Do ponto de vista teórico, a hierarquia é uma conveniência gráfica que não adiciona poder computacional, contudo permite uma modelagem compacta em níveis de abstração variados de sistemas de grande porte [20].

Nessa extensão, um conjunto de lugares e transições pode ser uma sub-rede da rede global. Para tanto, lugares e transições são utilizados como interface entre uma rede de mais alto nível e suas respectivas sub-redes.

Essa interface é vista como uma caixa-preta, ocultando detalhes do modelo representado. Quando a caixa-preta define a especialização de uma rede é chamada de subpágina; quando representa a generalização de uma rede denomina-se superpágina. Os elementos que se localizam nos extremos de uma subpágina são de mesmo tipo, e representam um lugar ou uma transição geral em uma superpágina. Por exemplo, caso busque-se especializar um evento, sua especificação deverá ter transições como entrada e saída da sub-rede.

Seguindo o exemplo das porcas e parafusos, a Figura 16 ilustra o uso de hierarquia com RdP. A parte superior da ilustração apresenta a RdP-C da Figura 15 em alto nível, onde os procedimentos de montar e enviar pacote estão encapsulados em uma única transição, representando a tarefa de empacotamento. Já a parte inferior mostra um refinamento, por meio da ligação da interface com a sub-rede.

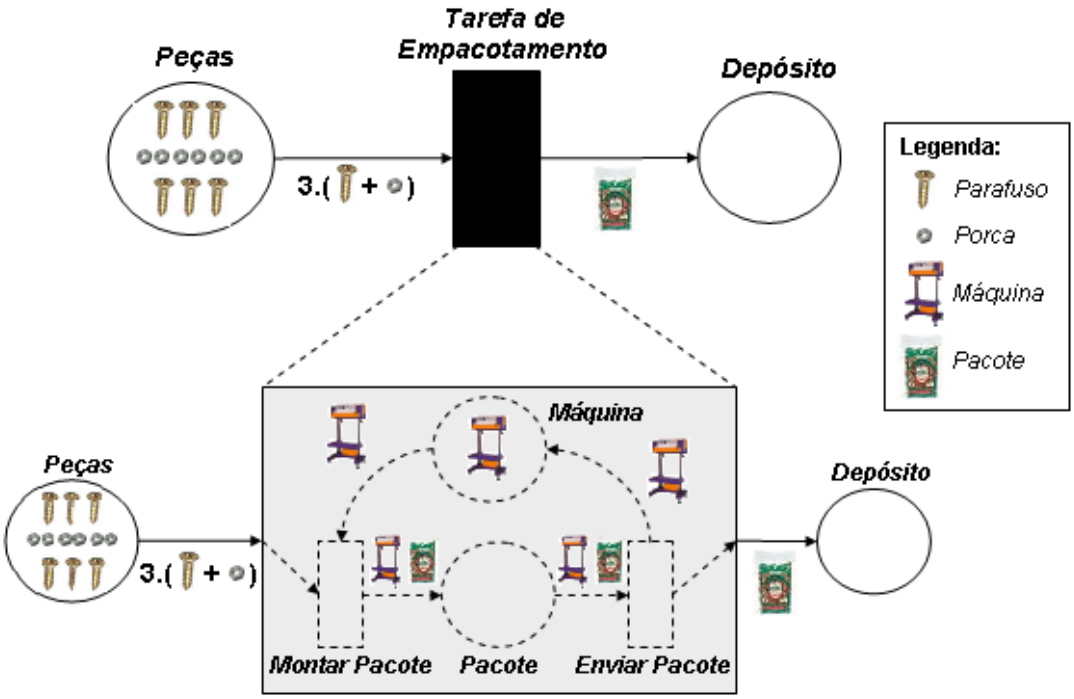


Figura 16 – Linha de produção simples representada por uma RdP-H.

3.2 Taxonomia de Decomposição de Tarefas

A análise do processo interativo está ligada à especificação das tarefas do ambiente. Cada tarefa a ser executada corresponde a um objetivo específico proposto pela aplicação. Apesar de AVs serem compostos por diferentes tipos de tarefas, é possível decompô-las em tarefas simples que formam a base do sistema de interação. Estas tarefas básicas geralmente envolvem técnicas de seleção, manipulação e navegação.

Por seleção entende-se o processo de definir, dentre os objetos de um AV, sobre qual ou quais deles se deseja manipular. Este processo envolve duas etapas: uma de indicação do objeto, onde o usuário "mostra" ao sistema o objeto de interesse, e outra de confirmação da seleção, onde o usuário define o objeto como selecionado.

Segundo Pinho [34], a manipulação consiste no processo de alteração de parâmetros ou estado de um objeto previamente selecionado. Esta mudança envolve a mudança da orientação, posicionamento, tamanho ou outro parâmetro qualquer, geométrico (forma ou posição), visual (cor ou textura) ou comportamental (iniciar movimento ou parar).

A navegação, por sua vez, é o processo que permite o deslocamento do usuário dentro do AV. Este processo pode envolver tanto mudanças de posição e rotação do avatar, como ferramentas que auxiliam o usuário a encontrar o caminho desejado e tarefas que controlam a velocidade do movimento. Para tanto, existem técnicas responsáveis pela locomoção e localização/ orientação (*wayfinding*) do usuário.

As tarefas acima descritas podem ser executadas através da utilização de **técnicas de in-**

teração em AVs. Baseado em Bowman [8], pode-se dizer que uma técnica de interação é um método que permite a concretização das tarefas em um AV de forma dinâmica, envolvendo tanto componentes de *hardware* como de *software*. A parte de *software* é responsável em mapear a informação de um dispositivo de entrada para alguma ação compreensível pelo sistema. Além disso, o resultado dessa operação é mapeado pela aplicação para um dispositivo de saída, que procurará representar as ações do usuário de forma tão natural quanto possível.

Pensando numa alternativa que permitisse a organização de técnicas de interação em conjuntos particulares, de maneira com que elas pudessem ser pensadas sistematicamente, Bowman [7] propôs uma taxonomia de técnicas de interação com base na decomposição das tarefas de uma aplicação. Neste conceito, cada tarefa de interação possa ser particionada (decomposta) em subtarefas simples e independentes. Similarmente, técnicas podem ser decompostas em subtécnicas, chamadas de componentes da técnica. Cada um destes componentes pode estar relacionado a uma subtarefa. A Figura 17 apresenta um esboço desta taxonomia de decomposição.

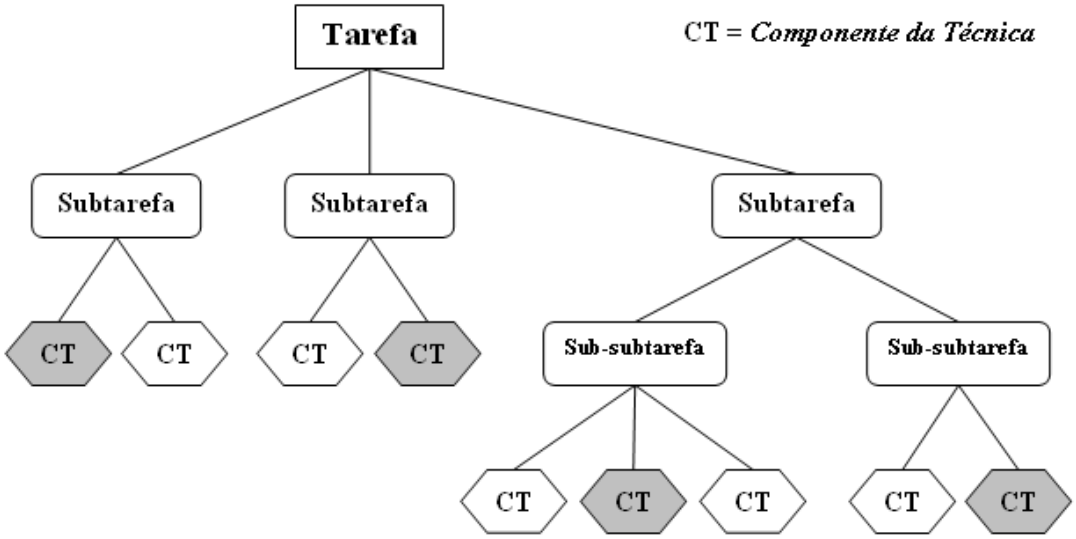


Figura 17 – Formato geral da taxonomia de decomposição de tarefas (adaptado de Bowman [8]). Os componentes da técnica em tom de cinza podem ser combinados para formar uma técnica de interação completa.

Para exemplificar o uso da taxonomia de decomposição de tarefas, suponha-se que uma tarefa básica seja alterar a cor de um objeto em um AV. Esta tarefa pode ser dividida, inicialmente, em três subtarefas diretamente relacionadas à tarefa: selecionar um objeto, escolher uma cor e aplicar esta cor. No entanto, podem existir também subtarefas que não estejam diretamente relacionados à tarefa do usuário, mas são igualmente importantes durante o processo de decomposição. A paleta de cores e os *feedbacks* de interação oferecidos pelo sistema, por exemplo, poderiam ser enquadrados neste caso.

Bowman [7] também destaca que uma taxonomia precisa ser suficientemente genérica para que técnicas de interação ajustem-se aos propósitos da tarefa. Por exemplo, a taxonomia poderia

oferecer três formas diferentes de interação para a tarefa de colorir um objeto: tocar o objeto, dar um comando de voz ou escolher um item de menu para aplicar a cor. No entanto, nada impede que exista uma outra técnica que desempenhe a mesma tarefa (uma forma de apontamento, por exemplo).

Taxonomia e categorização são boas formas para entender os detalhes de uma técnica de interação, e formalizar as diferenças entre elas [8]. E, uma vez formalizado, é possível determinar sua utilização durante o projeto do AV. Assim, a taxonomia serve como guia na avaliação de técnicas e de tarefas, ao invés de apenas identificar falhas de projeto.

Outra vantagem é a possibilidade de integrar componentes genéricos já desenvolvidos para a concepção de uma nova técnica. Este processo pode ser extremamente útil quando o número de subtarefas envolvidas num projeto de AV é pequeno, e a escolha para cada subtarefa seja clara o suficiente para permitir sua representação gráfica [7].

3.2.1 Taxonomia para técnicas de seleção e manipulação

Bowman apresenta uma taxonomia conjunta para técnicas de seleção e manipulação [7], uma vez que uma tarefa de manipulação requer, inicialmente, a execução de uma tarefa de seleção (o contrário não ocorre). Como visto anteriormente, seleção refere-se ao ato de especificar ou escolher um objeto para algum propósito. Já manipulação compreende a tarefa de ajustar a posição e orientação do objeto selecionado.

Nesta taxonomia, a tarefa de selecionar e manipular objetos é dividida em três tarefas básicas: seleção, manipulação e liberação, conforme mostra a Figura 18. Esta abordagem também permite que o projetista da técnica possa detalhar a tarefa, subdividindo-a novamente. Uma tarefa de manipulação poderia envolver, por exemplo, operações de anexação, posicionamento e orientação.

Baseado na taxonomia da Figura 18, é possível perceber que existem três (03) etapas distintas durante o processo de seleção, quatro (04) durante o processo de manipulação e duas (02) etapas durante o processo de liberação.

A primeira etapa do processo de seleção refere-se à **indicação** do objeto, onde o usuário "mostra" ao sistema qual objeto deseja manipular. A segunda diz respeito à **confirmação** do objeto, onde o usuário define o objeto como selecionado. No entanto, para que o usuário perceba suas ações no ambiente, tanto na indicação, como na confirmação do objeto selecionado, é necessário que o sistema sempre retorne um ***feedback*** apropriado a cada situação. Uma das alternativas utilizadas por aplicações é realçar o objeto selecionado com uma cor diferente, uma moldura ou marcadores. E, para o instante da confirmação, geralmente são empregados sinais visuais, sonoros ou táteis.

O processo de manipulação é iniciado pela etapa de **anexação** do objeto virtual à forma de apontamento do usuário ou a sua representação no AV. Após isto, tarefas que envolvem o **repo-**

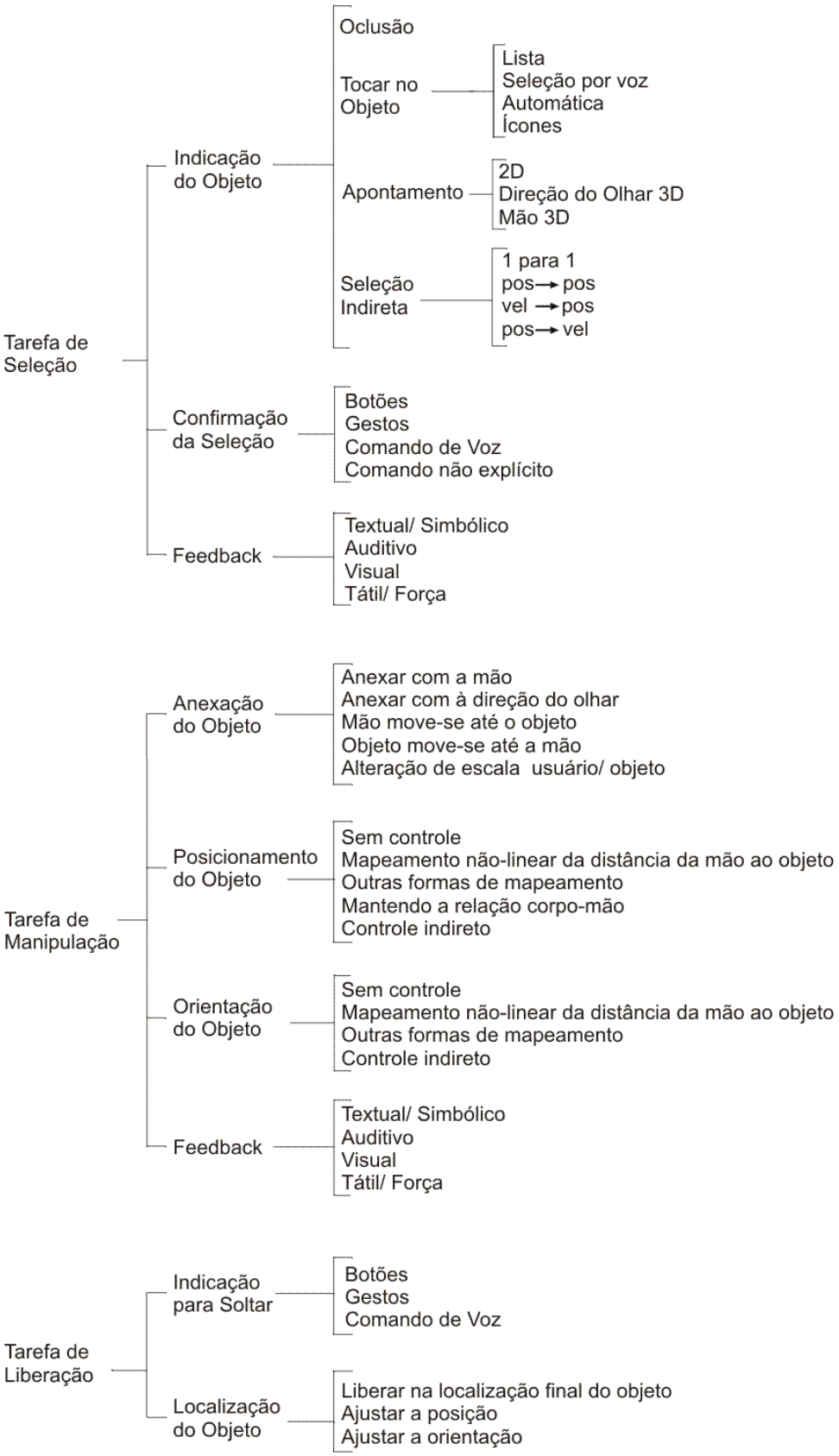


Figura 18 – Classificação de técnicas de seleção e manipulação pela decomposição de tarefas (adaptado de Bowman [7]).

sicionamento e **orientação** do objeto podem ser empregadas, em conjunto ou separadamente. Da mesma forma que uma tarefa de seleção, o sistema apresenta um ***feedback*** apropriado para que o usuário perceba as alterações no ambiente. Este *feedback* pode ser apresentado através do mapeamento instantâneo da ação do usuário.

O processo de liberação envolve, primeiramente, uma indicação de que o usuário está **liberando** o objeto em manipulação (através do pressionamento de um botão, gesto ou comando de voz). Após isto, o sistema apresenta um ***feedback*** que "solta" o objeto virtual em sua nova posição no AV.

Os componentes das técnicas listados na Figura 18, para cada uma das subtarefas apresentadas, podem ser combinados com o objetivo de produzir uma técnica de interação completa. A capacidade de combinar componentes em novas maneiras já tem resultados expressivos em trabalhos apresentados pela comunidade de RV. As técnicas HOMER [6] e Voodoo Dools [33] são exemplos que combinam diferentes componentes para a realização de tarefas de seleção e manipulação.

3.2.2 Taxonomia para técnicas de navegação

Conforme destacado anteriormente, a navegação refere-se à forma como o usuário movimenta-se dentro do cenário virtual. Esta tarefa envolve técnicas de deslocamento e orientação (*way-finding*), sendo que a primeira consiste no componente motor da navegação virtual, relacionado as tarefas físicas realizadas no mundo real, enquanto que a segunda refere-se ao componente cognitivo em um processo de navegação: pensamento, planejamento e decisão do caminho e dos movimentos a serem executados.

A taxonomia para técnicas de navegação decompõe a tarefa básica de navegação em três subtarefas: seleção da direção e do alvo, seleção da velocidade e aceleração e as condições de entrada, conforme mostra a Figura 19.

A **seleção da direção e do alvo** refere-se à subtarefa na qual o usuário especifica como quer mover-se e para onde quer mover-se. Quando a direção é determinada pelo movimento da mão, por exemplo, o usuário sempre se desloca para onde sua mão ou dedo estiver apontando. Um fator de escala também pode ser utilizado como instrumento de navegação, facilitando o apontamento de objetos-alvo a serem alcançados. Outras formas de navegação utilizam o controle da direção pelo movimento da cabeça e o uso de dispositivos físicos, como *joysticks*.

Uma forma fácil de controlar a **velocidade e a aceleração** do movimento em AVs é torná-las constantes. No entanto, isto pode não ser muito vantajoso quando uma rota distante precisa ser percorrida. Para estes casos, técnicas que permitam movimentar-se mais rapidamente enquanto se está longe do destino, diminuindo gradativamente a velocidade à medida que se aproxima, e controlando a aceleração por meio de gestos ou comandos de voz são alternativas interessantes.

As **condições de entrada** controlam o período de navegação do usuário. Por exemplo, o

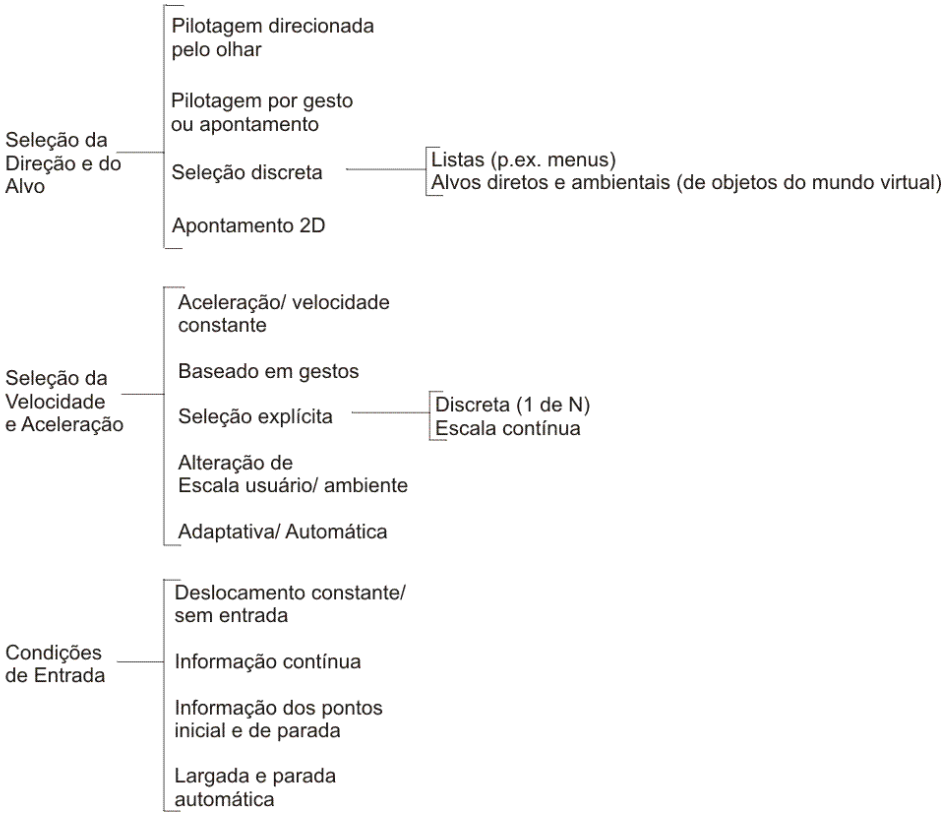


Figura 19 – Classificação de técnicas de navegação pela decomposição de tarefas (adaptado de Bowman [7]).

usuário indica ao ambiente, através de um dispositivo de entrada, quando um deslocamento é iniciado ou terminado. Este deslocamento também pode sofrer interrupções para descanso ou observação de sinais de indicação presentes no AV.

4 Base da Metodologia

O objetivo principal desta metodologia é modelar e implementar componentes que representem as etapas do processo interativo. O emprego de um formalismo, em conjunto com uma taxonomia de interação, possibilita a especificação detalhada do sistema, bem como auxilia o processo de estruturação e implementação, permitindo o encapsulamento de determinadas funcionalidades. Tais características podem permitir que componentes gerados sejam reutilizados, simplificando o processo de desenvolvimento e adicionando flexibilidade às aplicações.

Para tanto, este Capítulo apresenta a base da metodologia proposta, onde o Formalismo de RdP e a Taxonomia de Decomposição de Tarefas são integrados para a modelagem e implementação de TIs. Esta integração permite a descrição de uma tarefa de interação, onde a seqüência de passos a serem desenvolvidas pelo usuário é controlada por uma RdP.

4.1 Justificativa

Para a base da metodologia, optou-se pelo emprego de RdP por esta apresentar um formato de especificação formal com diferentes formas de extensão, bastante fundamentado e consolidado pela comunidade científica [54]. Conforme Murata [23], uma RdP é uma ferramenta de modelagem gráfica e matemática para especificação e análise de sistemas concorrentes e dinâmicos, onde aplicações de RV se enquadram.

Além disso, o poder de expressão de seu formalismo, juntamente com as ferramentas de análise e simulação existentes, permitem que a estrutura e o comportamento de um sistema sejam previstos e testados antes da fase de implementação.

A representação gráfica adotada para a metodologia baseia-se no uso de RdP-C e RdP-H, como forma de distinguir os diferentes tipos de dados a serem manipulados por uma aplicação e permitir a hierarquização dos modelos. Conforme visto anteriormente, ambas abordagens são uma extensão das RdP que têm por objetivo reduzir o tamanho dos modelos. Por convenção, este trabalho irá referir-se à RdP-C pelo simples termo de “RdP”, enquanto que “RdP-H” fará menção às Redes de Petri Coloridas Hierárquicas.

A proposta também procura aproximar a concepção do usuário ao trabalho do projetista e do desenvolvedor, modelando a aplicação sob a perspectiva das tarefas que o usuário deve desempenhar no AV. Estas tarefas podem ser agrupadas em “tarefas-base” que, quando organizadas por características similares, procuram definir as etapas do processo intera-

tivo. Conforme já mencionado, Bowman [7] propõe uma taxonomia para interação que prevê a divisão deste processo em três tarefas elementares, seguindo esta ordem: **seleção**, **manipulação** e **liberação**.

Este trabalho procura adaptar essa taxonomia, separando a tarefa de seleção em duas etapas distintas. A primeira, denominada **seleção**, é responsável pelo processo de indicação do objeto que se deseja manipular. Já a segunda, denominada **anexação**, trata do processo de confirmação da seleção. Ambas apresentam *feedbacks* que informam o usuário sobre sua execução. As tarefas de manipulação (posicionamento e orientação) e liberação permanecem inalteradas, seguindo a concepção original de Bowman.

Com base nessas definições, a metodologia de modelagem desenvolvida neste trabalho prevê que cada elemento de uma RdP (**lugares**, **transições**, **arcos** e **marcas**) represente um determinado papel durante o processo interativo de uma aplicação de RV.

4.2 Definição de Papéis

No modelo desenvolvido, **Lugares** são elementos que **determinam o estado atual** da interação. Eles dispõem de canais de comunicação, capazes apenas de receber e transmitir informações necessárias para o funcionamento da rede, não produzindo dados.

Para que uma tarefa de seleção esteja disponível ao usuário, por exemplo, é necessário que o **lugar** que representa o estado de seleção contenha dados que indiquem, ao menos, a forma de apontamento, os objetos disponíveis para seleção e que nenhum objeto está sendo manipulado neste momento. A Figura 20 representa esta situação usando um **lugar** e uma **transição**.

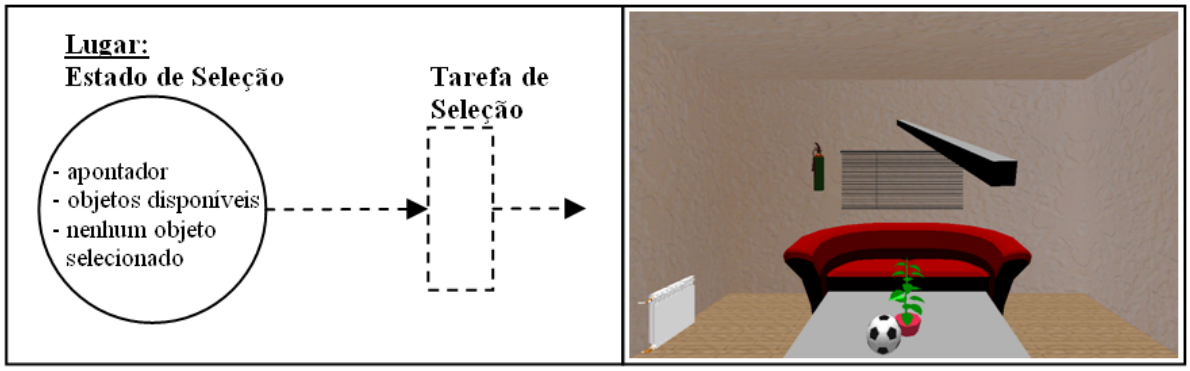


Figura 20 – O *Estado de Seleção* fornece as informações necessárias que possibilitam ao usuário executar a *Tarefa de Seleção*.

Transições são elementos que **realizam o processamento**, alterando o comportamento da aplicação. Elas representam, em alto nível, as **tarefas elementares de interação**. Assim como os **lugares**, as **transições** dispõem de canais de comunicação para a recepção e transmissão de informações pela rede. No entanto, elas podem consumir dados, bem como gerar e inserir novos dados na rede.

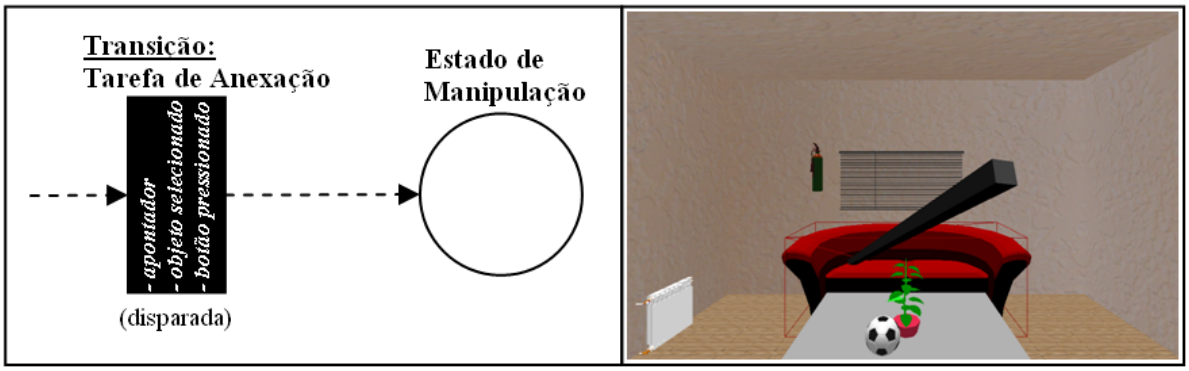


Figura 21 – Exemplo de **transição** que executa a anexação de um objeto selecionado para um apontador. Após esta tarefa, a aplicação passa para o *Estado de Manipulação*.

Um exemplo de seu funcionamento pode ser representado pelo exato momento em que o usuário confirma a seleção de um objeto. A **transição** responsável por esta tarefa é então disparada, executando a operação que anexa o objeto ao apontador do usuário. A Figura 21 representa este exemplo.

Arcos determinam a **ordem de execução da rede**. Eles são responsáveis pelo **transporte de dados** entre um **lugar** e uma **transição** (e vice-versa), indicando pré e pós-condições para a execução das **transições** ou para o estabelecimento de um estado. Conseqüentemente, os **arcos** definem a ordem de execução das tarefas no AV.

Para manipular um objeto, por exemplo, é necessário antes executar tarefas de seleção e anexação que fornecem informação sobre o objeto escolhido pelo usuário. A Figura 22 apresenta esta seqüência, rotulando os **arcos** com os dados necessários para a realização de cada tarefa.

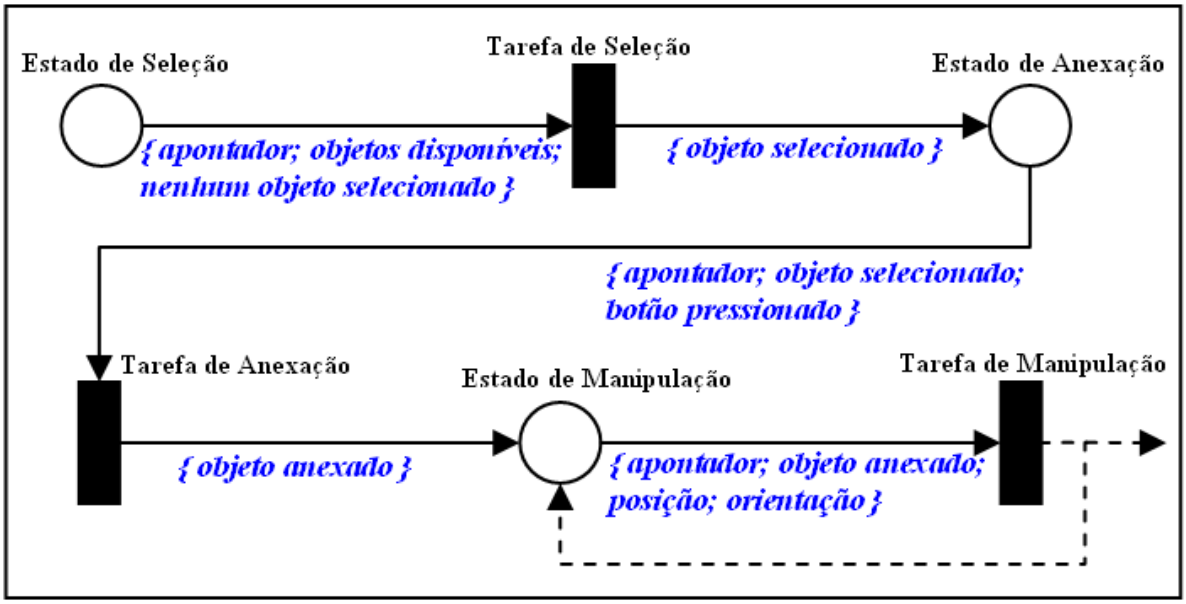


Figura 22 – **Arcos** entre **lugares** e **transições** definindo a ordem de execução e os recursos de cada etapa do processo interativo.

Marcas representam os **recursos disponíveis** para o funcionamento da aplicação, bem como da RdP. Dados como objetos geométricos, menus, vetores de posicionamento e cliques de botões são exemplos de possíveis **marcas**. Tais recursos são provenientes da própria aplicação ou de dispositivos de entrada (como *mouse*, teclado, *joystick* ou rastreador), podendo ser armazenados em **lugares** da rede e terem seus conteúdos atualizados por **transições**. Nesta metodologia, as **marcas** distinguem-se umas das outras pelo tipo de dado que elas encapsulam. A Figura 23 apresenta um exemplo onde os recursos de uma aplicação são representados na RdP na forma de ícones.

Por ser baseado em RdP, a metodologia também exige a definição de **marcas iniciais**. Sob o ponto de vista de uma aplicação, estas **marcas** referem-se à **configuração inicial do sistema**. Dados vindos dos dispositivos de entrada e a lista dos objetos geométricos que compõem um AV (fornecidos pela aplicação) podem ser considerados exemplos de **marcas iniciais**, e precisam ser fornecidos para algum **lugar** da RdP. Seguindo o formalismo, deve se especificar ao menos um **lugar** que guarde estas **marcas iniciais**.

Figura 23 – Objetos e sua representação como marcas numa RdP.

De acordo com as regras de formação das RdP, cabe lembrar que um nodo (**lugar** ou **transição**) não pode ser ligado diretamente a outro nodo de mesmo tipo. Portanto, sempre entre dois **lugares** haverá uma **transição**, da mesma forma que entre duas **transições** haverá um **lugar**.

Como forma de sintetizar o que foi apresentado nesta Seção, a Tabela 3 apresenta um resumo do papel que cada elemento da RdP representa durante a especificação do processo interativo. Estes conceitos são fundamentais para o emprego correto da metodologia, assunto do próximo Capítulo.

Tabela 3 – RdP e suas funções durante o processo interativo.

Elementos da RdP	**Papel no Processo Interativo**
Lugares	- Determinam o estado atual de interação - Podem armazenar os recursos disponíveis
Transições	- Desempenham as ações, alterando o comportamento da aplicação - Representam as tarefas elementares de interação
Arcos	- Determinam a ordem de execução das tarefas - Indicam pré e pós-condições de uma ação
Marcas	- Representam os recursos disponíveis - Marcas iniciais definem a configuração inicial do sistema

5 Empregando a Metodologia

A partir dos conceitos apresentados nos Capítulos anteriores, já é possível representar as etapas do processo interativo de uma aplicação de RV utilizando a metodologia proposta.

Para tanto, este Capítulo apresenta um exemplo do uso da metodologia. Um AV que envolve tarefas de seleção e manipulação foi escolhido para servir como plataforma de testes. Sua descrição permite que as etapas do processo interativo sejam identificadas, modeladas e codificadas. Testes com o modelo procurando observar o comportamento da aplicação também são apresentados, bem como uma forma de representação hierárquica do sistema.

5.1 Plataforma de Teste

Para ilustrar o uso desta metodologia na especificação do processo interativo em AVs, foi utilizada uma aplicação de Quebra-Cabeça Virtual [40], onde o objetivo principal do usuário é escolher e encaixar corretamente cada peça do jogo, realizando para tal tarefas de seleção e manipulação de objetos.

A Figura 24 apresenta as características do cenário virtual. Inicialmente, as peças do quebra-cabeça localizam-se misturadas ao lado direito da área de visualização e a caixa para montagem localiza-se à esquerda. A ordem de escolha dos blocos não é um quesito levado em consideração pela aplicação. Para interagir, usa-se a técnica de mão virtual.

Esta aplicação foi construída utilizando a linguagem C++ e as bibliotecas OpenGL, GLUT e SmallVR [35] [36]. Esta última é uma biblioteca gráfica que facilita o desenvolvimento de aplicações de RV, abstraindo diferentes aspectos de implementação como o controle de dispositivos e o gerenciamento de um grafo de cena, não perdendo a estrutura básica de um programa construído com a GLUT. A SmallVR traz suporte a dispositivos convencionais, como teclado, *mouse* e monitor e não-convencionais, como rastreadores e óculos de RV.

Para o emprego da metodologia, quatro passos devem ser seguidos pelo projetista da aplicação, quais sejam:

- Identificar as **etapas** do processo interativo (**tarefas** do AV), de acordo com a taxonomia de decomposição de Bowman, bem como os **estados** assumidos após a execução de cada tarefa;

- Definir uma RdP com **tarefas** e **estados** identificados pelo passo anterior;

- Identificar os **recursos** necessários para cada etapa do processo interativo (prés e pós-condições), adicionando **marcas** aos **arcos** da RdP do passo anterior;
- Implementar o modelo, utilizando um conjunto de classes especialmente desenvolvido para construir a RdP e controlar sua execução.

As próximas seções apresentam o detalhamento de cada um destes passos.

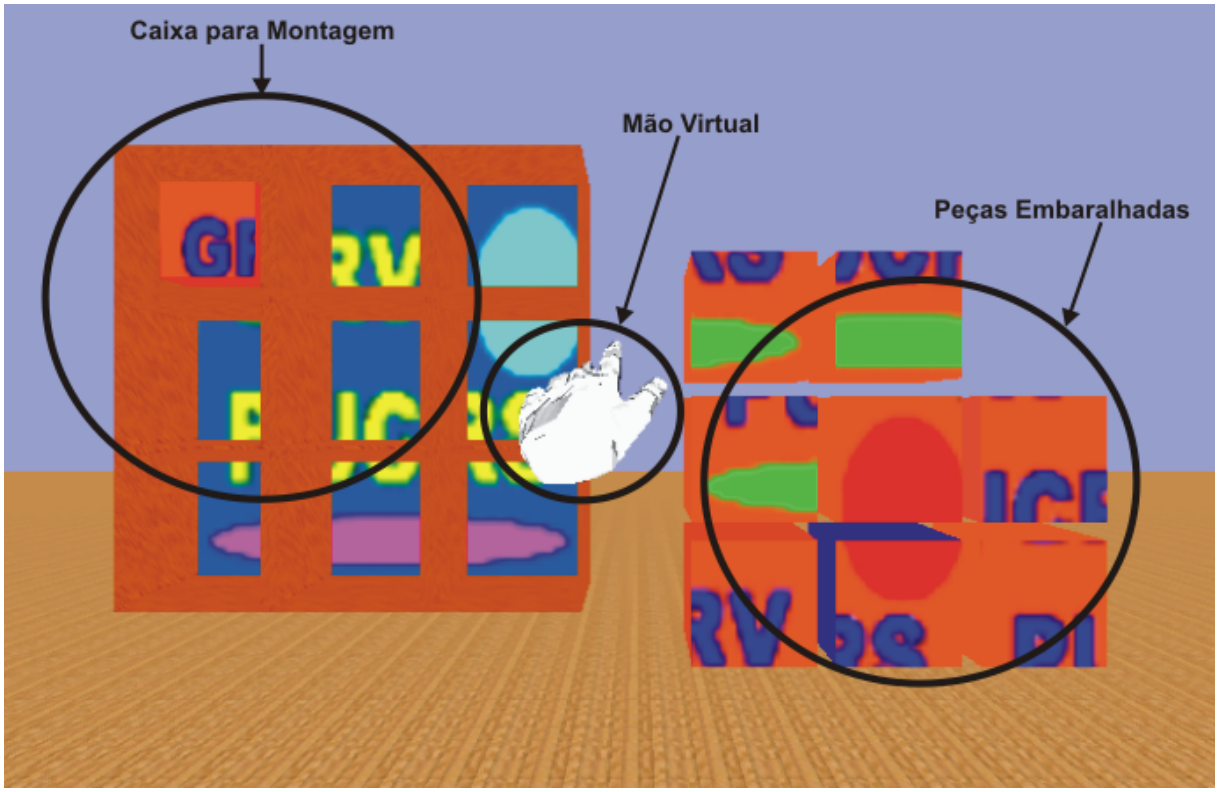


Figura 24 – Quebra-Cabeça Virtual usando a técnica de mão virtual.

5.2 Identificando as Etapas do Processo Interativo

Analisando a aplicação sob a perspectiva da taxonomia apresentada no Capítulo 4, que divide a tarefa de interação em seleção, anexação, manipulação e liberação, a identificação das tarefas e estados assumidos pela aplicação é feita a seguir (sugere-se acompanhar o texto, observando a Figura 25).

A aplicação inicia no *Estado de Seleção*, no qual o usuário move um apontador (mão virtual) no ambiente, procurando por um objeto para seleção. A partir deste estado, a *Tarefa de Seleção* realiza um teste de colisão entre o apontador e as peças do quebra-cabeça. Se a colisão existir (um objeto sendo apontado), o *Estado de Anexação* é habilitado.

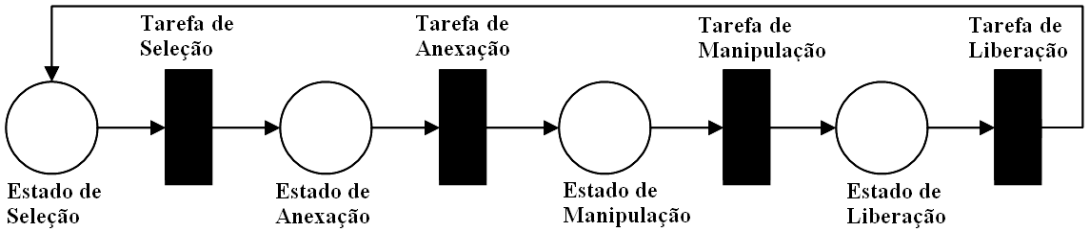


Figura 25 – RdP em alto nível representando as quatro tarefas básicas de interação.

A partir deste ponto, se o usuário pressionar (e não soltar) o botão de seleção, a *Tarefa de Anexação* é disparada. Esta tarefa "cola" o objeto ao apontador, estabelecendo o *Estado de Manipulação*.

Uma vez estabelecido este estado, a *Tarefa de Manipulação* é disparada, permitindo que o usuário reposicione o objeto usando a mão virtual. A localização do objeto tem por base a posição da mão na qual o mesmo está anexado.

Para simplificar a especificação, neste exemplo, algumas operações de *feedback* normalmente fornecidas ao usuário foram omitidas.

Enquanto a peça está sendo movimentada, uma ação de "soltar" o objeto pode ser realizada. Se o botão de seleção for solto, o *Estado de Liberação* é habilitado, o qual imediatamente dispara a *Tarefa de Liberação* que separa o objeto do apontador, retornando a aplicação ao *Estado de Seleção*.

5.3 Definindo um Modelo de RdP

Após identificadas as tarefas da aplicação em um alto nível de abstração, procede-se o mapeamento que divide as tarefas em partes menores, baseando-se nas operações que cada uma desempenha [7]. A Tabela 4 apresenta a decomposição das tarefas elementares, enquanto a Figura 26 mostra a nova configuração da RdP.

Tabela 4 – Detalhamento as quatro tarefas básicas de interação.

Tarefas de Alto Nível	**Operações Básicas**
Tarefa de Seleção	- Subtarefa de Indicação - Subtarefa de *Feedback* de Indicação
Tarefa de Anexação	- Subtarefa de Confirmação - Subtarefa de *Feedback* de Confirmação
Tarefa de Manipulação	- Subtarefa de Posicionamento
Tarefa de Liberação	- Subtarefa de Separação - Subtarefa de *Feedback* de Separação

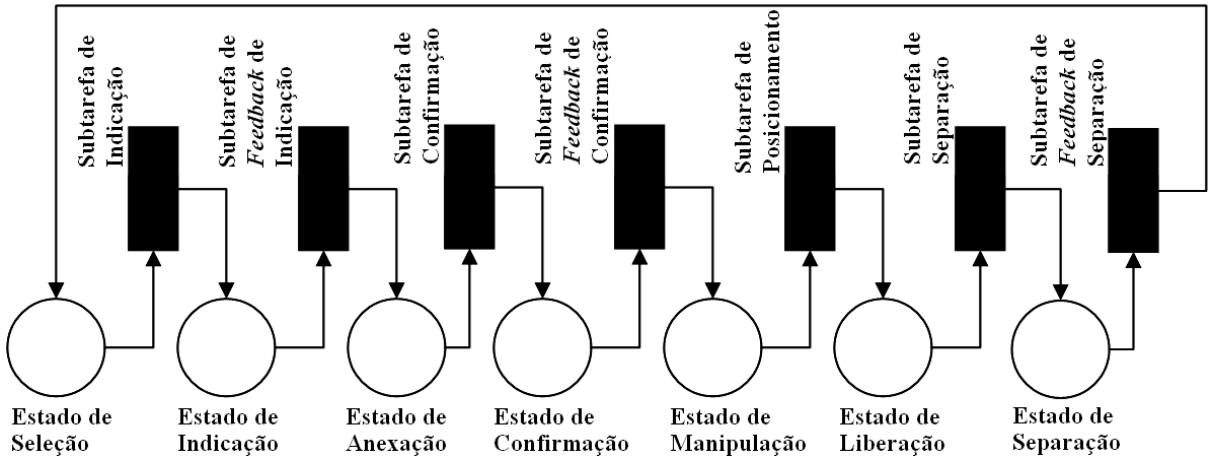


Figura 26 – RdP representando o detalhamento do processo interativo da aplicação de Quebra-Cabeça Virtual.

5.4 Identificando os Recursos para o Processo Interativo

Seguindo a metodologia, passa-se a identificar os recursos necessários (dados) para cada etapa do processo interativo, representando-os na RdP como **marcas**. Para tanto, os **arcos** da rede devem ser rotulados com estas **marcas** que, neste caso, são representadas por ícones.

Os **lugares** *Estado de Seleção*, *Estado de Anexação*, *Estado de Manipulação* e *Estado de Liberação* necessitam constantemente de informações sobre o estado dos dispositivos e de variáveis de controle da aplicação. Para tanto, **marcas** referentes a estes dados são depositadas nestes **lugares**. Neste exemplo, tais **marcas** referem-se à forma de apontamento, à lista de objetos do cenário e aos dados provenientes dos dispositivos utilizados.

O modelo completo da RdP que representa o processo interativo do Quebra-Cabeça Virtual é apresentado na Figura 27, na qual os dispositivos são representados por um triângulo, enquanto a aplicação é representada por um hexágono. Estas formas são meramente ilustrativas e servem apenas para facilitar a compreensão visual do funcionamento da rede.

A execução de um passo da rede consiste na verificação da existência das pré-condições de cada transição, disparando aquelas que estiverem habilitadas a cada ciclo de *rendering*.

De maneira resumida, um ciclo completo de execução da RdP da Figura 27 pode ser analisado e interpretado da seguinte forma: inicialmente, a aplicação e o dispositivo enviam **marcas** para os **lugares** *Estado de Seleção*, *Estado de Anexação*, *Estado de Manipulação* e *Estado de Liberação*. Como a **transição** *Subtarefa de Indicação* recebe todas as **marcas** necessárias do *Estado de Seleção*, esta é imediatamente disparada, coletando seus dados (a lista de objetos do cenário, o apontador e uma variável indicando que não existe objeto selecionado). Esta **transição** realiza um teste de colisão entre o apontador do usuário e os objetos do cenário. Havendo colisão com algum objeto, uma nova **marca** é gerada, representando este objeto.

Transcorrida esta etapa, a aplicação passa para o *Estado de Indicação* que, uma vez esta-

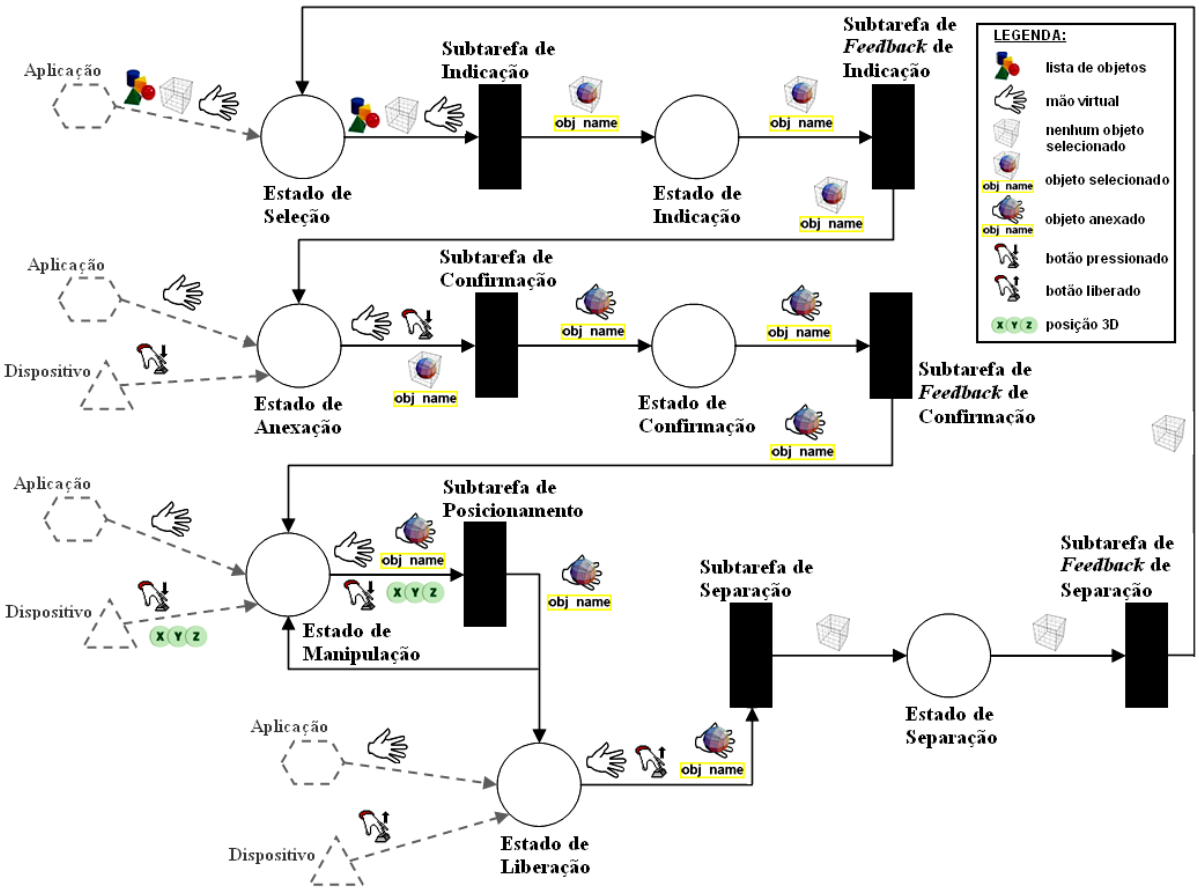


Figura 27 – RdP com as **marcas** necessárias para habilitar as **transições** e depositar dados nos **lugares** da rede que representam o processo interativo do Quebra-Cabeça Virtual.

belecido, dispara a **transição** *Subtarefa de Feedback de Indicação* responsável por aplicar um realce sobre este objeto, de maneira a destacá-lo dos demais objetos do AV.

Em seguida, o **lugar** *Estado de Anexação* recebe a **marca** que representa o objeto selecionado. Neste estado, ele já dispõe das **marcas** referentes ao apontador utilizado e de um possível botão pressionado pelo usuário. Quando todos estes dados estiverem presentes neste **lugar**, a **transição** *Subtarefa de Confirmação* é disparada "colando" o objeto selecionado ao apontador do usuário.

A seguir, a **marca** que representa o objeto selecionado é transmitida para o *Estado de Confirmação*. Nestas condições, a *Subtarefa de Feedback de Confirmação* é disparada, emitindo um sinal sonoro comunicando o sucesso no processo de anexação. Após isto, uma **marca** que encapsula o nome do objeto selecionado é transmitida para o *Estado de Manipulação*, que representa o início do processo de manipulação, onde o usuário pode posicionar corretamente o objeto na caixa de montagem. Em face da configuração da rede, este estado sempre dispõe das **marcas** que representam o apontador (recebido da aplicação), dos dados que chegam do dispositivo de rastreamento e do botão que está sendo pressionado pelo usuário. Assim, com a chegada da **marca** do objeto selecionado a **transição** *Subtarefa de Posicionamento* é disparada. Esta **transição** atualiza a posição do objeto, baseada nos dados do rastreador, gerando

e enviando uma nova **marca** com o objeto selecionado, para o *Estado de Liberação* e para o *Estado de Manipulação*. Enquanto o botão de seleção estiver sendo pressionado, a **transição** *Subtarefa de Posicionamento* é repetidamente disparada, permitindo ao usuário reposicionar o objeto tantas vezes quanto necessárias.

Se o botão for liberado, o *Estado de Manipulação* não oferecerá mais as pré-condições necessárias para a *Subtarefa de Posicionamento* ser disparada. Ao mesmo tempo, o *Estado de Liberação* recebe uma **marca** informando que o usuário soltou o botão, além das **marcas** que representam o objeto selecionado e o apontador utilizado. Nestas circunstâncias, a **transição** *Subtarefa de Separação* é disparada. Esta **transição** "solta" o objeto no AV em sua nova posição. Seqüencialmente, o *Estado de Separação* recebe uma **marca** que dispara a **transição** *Subtarefa de Feedback de Separação*, que emite um sinal sonoro comunicando que o processo de liberação foi realizado com sucesso. Uma **marca** é então repassada para o *Estado de Seleção*, designando que uma nova seleção já pode ser executada.

5.5 Implementando o Modelo de RdP

Nesta Seção primeiramente são apresentados as classes que permitem a criação da RdP. Em seguida, uma breve explicação sobre a comunicação entre lugares e transições é apresentada. Após isto, é mostrada a metodologia para geração de código necessário à execução da RdP.

5.5.1 Classes que Representam a RdP

Uma vez concluída a modelagem, é possível iniciar o procedimento de implementação. Com o objetivo de automatizar o processo de geração de código, foi desenvolvido um conjunto de classes na linguagem C++, descrito pelo diagrama UML apresentado na Figura 28. Estas classes representam os elementos da RdP, permitindo a implementação dos modelos que representam as tarefas de interação de uma aplicação. Com elas, é possível simular e controlar o fluxo de atividades do sistema.

Note-se que todas as classes do diagrama herdam as características da classe *Object*, responsável por armazenar o nome e tipo de cada objeto. Outra classe importante neste modelo é *InteractObj*. Ela define que todas as suas classes derivadas implementem um método *run*, responsável pelo processamento de elemento durante a execução da RdP.

ConectorIn e *ConectorOut* são as classes que representam os canais de comunicação de **lugares** e **transições**, e têm por função guardar em cada elemento os dados trafegados pela RdP. Já *Token* é a classe responsável por encapsular os vários tipos de dados que o desenvolvedor utiliza na aplicação.

PetriNet é a classe responsável por simular e controlar a rede. Ela possui uma lista que pode

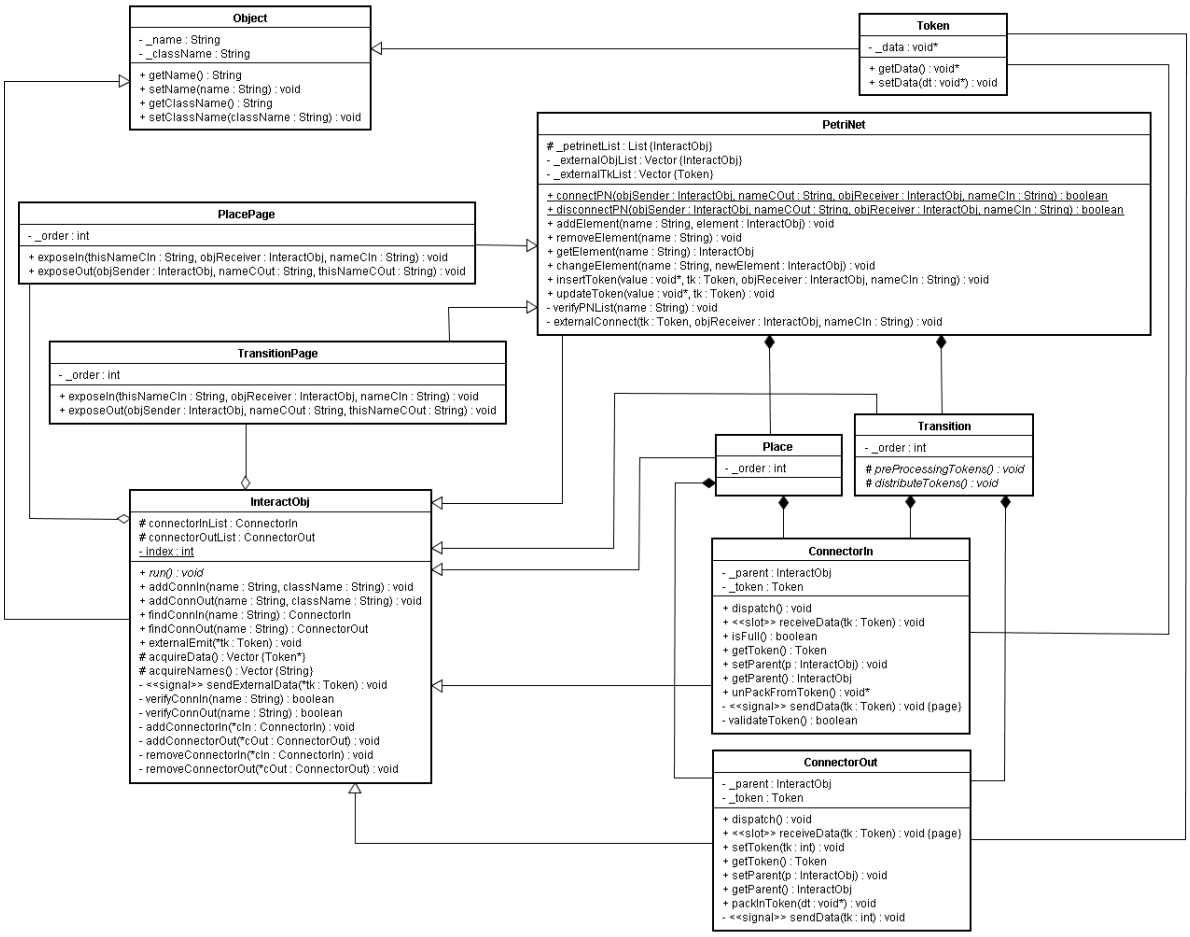


Figura 28 – Conjunto de classes usado para implementer o modelo de RdP.

armazenar os nodos que compõe a RdP.

As classes *Place* e *Transition* representam **lugares** e **transições**, respectivamente, e podem ter associados a si canais de comunicação. Para casos que precisam representar sub-redes, dois tipos especiais de classes podem ser utilizadas: *PlacePNPage* e *TransitionPNPage*.

Maiores detalhes referentes ao conjunto de classes apresentado (descrição de métodos e atributos) podem ser vistos no Manual do Usuário, presente no Apêndice B deste trabalho.

5.5.2 Comunicação entre Lugares e Transições

Todas as classes apresentadas têm como base o mecanismo de *signals* e *slots* [47] para a comunicação entre **lugares** e **transições**. Conforme Trolltech [47], *signals* são notificações (mensagens) de um objeto para outro que sinalizam a ocorrência de um determinado evento, enquanto *slots* são respostas do objeto receptor a estas mensagens. Sob o paradigma de orientação a objeto, esta resposta é um método a ser executado. Neste projeto, este mecanismo foi implementado usando a *classe QObject* do *toolkit* Qt [47]. Com este recurso, **lugares** e **transições**

podem ser implementados como objetos interconectados, permitindo a passagem das **marcas** durante a simulação da RdP.

Para exemplificar este mecanismo, imagine-se um AV para o treinamento de motoristas [3]. Um objeto *carro* possui uma função (*slot*) que ajusta seu ângulo de direção, e que precisa de um valor de entrada para operar. Qualquer dispositivo pode ser usado para dirigir o veículo, desde que este forneça um dado (*signal*) que satisfaça a função do objeto carro. Assim, basta existir uma comunicação entre o objeto que detém o dado, emitido pelo dispositivo, e o objeto *carro* que possui a função que calcula a direção.

5.5.3 Processo de Geração de Código

As classes que definem **lugares** e **marcas** podem ser diretamente usadas para instanciar objetos. **Arcos** são representados pelo uso explícito do método *connectPN* da classe *PetriNet*. Já **transições** devem ser implementadas como novas classes derivadas da classe abstrata *Transition*, que representa uma **transição** genérica. Por ser uma classe abstrata, esta abordagem força o desenvolvedor a codificar alguns métodos essenciais para a simulação do modelo, bem como permite o reuso de classes já desenvolvidas e utilizadas em projetos anteriores.

O reuso de componentes é útil quando procura-se verificar qual técnica de interação melhor se adapta a um AV. Componentes das técnicas que representam *feedbacks*, por exemplo, poderiam ser freqüentemente testados e trocados durante o projeto a fim de avaliar qual deles proporciona ao usuário a melhor percepção do processo interativo.

Para começar a implementação do modelo o projetista deve instanciar um objeto da *classe PetriNet* (veja Figura 29, linha 01). Este objeto representa a RdP como um todo e é usado para avançar a simulação da rede e para armazenar **lugares** e **transições**. Além disso, dispõe de métodos que encapsulam e recuperam os conteúdos das **marcas** e realizam a conexão entre os nodos da RdP.

Após o instanciamento do objeto que representa a RdP, criam-se os **lugares** e **transições** (veja Figura 29, linhas 02 e 03), além das **marcas** que circularão pela rede (veja Figura 29, linha 04). **Lugares** devem ser instanciados a partir da *classe Place*, enquanto **transições** são instanciadas a partir de classes derivadas da *classe Transition* (por exemplo, a *classe Indication* da Figura 29), e **marcas** a partir da *classe Token*.

Posteriormente, **lugares** e **transições** são adicionados à lista do objeto **rede** (veja Figura 29, linhas 05 e 06). Para tanto, o método *addElement* é utilizado, recebendo como parâmetro o nome atribuído ao elemento pela aplicação e o objeto que o representa.

Conforme apresentado no Capítulo 4, **lugares** e **transições** trocam **marcas** através de canais de comunicação. Para estabelecer uma ligação entre estes canais, "conectores" devem ser criados e adicionados aos **lugares** e **transições**. Cada conector é identificado por um nome e pelo tipo de dado que irá suportar (veja Figura 29, linhas 07 e 08). A classe *InteractObj* oferece os

Linha	Código
1	`PetriNet *pn = new PetriNet();`
2	`Place *pSel = new Place();`
3	`Indication *tInd = new Indication();`
4	`Token *tkObjs = new Token();`
5	`pn->addElement("Estado de Seleção", pSel);`
6	`pn->addElement("Subtarefa de Indicação", tInd);`
7	`pSel->addConnOut("objs", "vector");`
8	`tInd->addConnIn("objs", "vector");`
9	`pn->connectPN(pSel, "objs", tInd, "objs");`
10	`pn->insertToken(vecObjs, tkObjs, pSel, "objs");`
11	`pn->run();`

Figura 29 – Um simples exemplo de código gerado na fase de implementação.

recursos para criação e manipulação desses objetos. A Figura 30 ilustra um pequeno exemplo do uso de conectores em elementos de uma RdP.

Com a declaração de todos os conectores, a comunicação entre dois ou mais objetos pode ser estabelecida. Para tanto, o projetista deve explicitamente conectá-los, definindo o objeto emissor e seu conector, bem como o objeto receptor e respectivo conector (veja Figura 29, linha 09). O método *connectPN* deve ser utilizado para que este procedimento seja realizado.

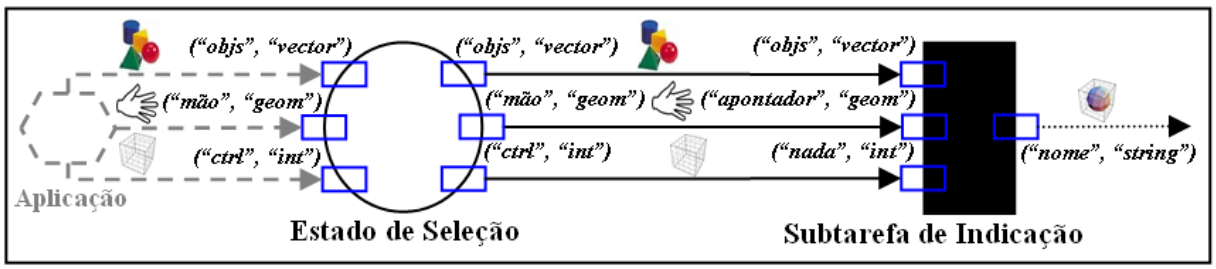


Figura 30 – Conectores de cada elemento da RdP. A única restrição para a ligação entre conectores é que o tipo de dado definido seja o mesmo para ambos.

No caso específico das **transições**, é preciso instanciar objetos de classes derivadas da *classe Transition* que representem as tarefas do processo interativo simbolizadas no modelo. Para o exemplo apresentado pela Figura 26, foram identificadas e criadas as seguintes classes, bem como definidas as suas funcionalidades:

- ***Indication***: detecta a colisão entre objetos do AV com o apontador do usuário;
- ***Indication_Feedback***: aplica um realce no objeto selecionado;

- ***Confirmation***: anexa o objeto selecionado ao apontador;
- ***Confirmation_Feedback***: notifica a anexação;
- ***Positioning***: atualiza a posição do objeto em manipulação;
- ***Detachment***: libera o objeto do apontador;
- ***Detachment_Feedback***: notifica a liberação.

De forma semelhante aos filtros definidos por Figueroa [11], a interface da classe abstrata *Transition*, que dá origem as classes derivadas de **transições**, obriga o desenvolvedor a implementar três métodos virtuais responsáveis pela coleta (*preProcessingTokens*) e distribuição (*distributeTokens*) de **marcas** e pelo processamento dos dados (*run*) dentro das **transições**.

O método *preProcessingTokens* recebe **marcas**, efetua o desempacotamento destas, e converte-as para tipos de dados aceitos pela aplicação. Já o método *run* permite ao desenvolvedor inserir o código específico da aplicação, realizando o que for necessário para o funcionamento do sistema. Neste estágio, dados podem ter seus valores atualizados, bem como novos dados podem ser gerados. Finalmente, o método *distributeTokens* realiza o empacotamento dos dados da aplicação para dentro de **marcas**, as quais são enviadas para a RdP.

Tome-se como exemplo a criação da classe *Indication_Feedback*, que representa a transição *Subtarefa de Feedback de Indicação* da Figura 27 que tem por função aplicar um realce ao objeto selecionado.

O método *run* desta classe, primeiramente, cria um objeto geométrico temporário para armazenar um objeto selecionado (veja Figura 31). A seguir, uma chamada do método *preProcessingTokens* é efetuada, a fim de desempacotar o objeto selecionado encapsulado pela **marca** da RdP (veja Figura 32). Existindo um objeto selecionado, um realce (borda) é desenhado ao seu redor. Finalmente, uma chamada ao método *distributeTokens()* é realizada, encapsulando as novas propriedades do objeto selecionado em uma **marca** da RdP (veja Figura 33).

```
objGeometrico *objSel;

void Indication_Feedback::run() {
   preProcessingTokens();        // faz a coleta e obtém as marcas
   if (objSel) objSel->drawBB(); // desenha um realce de envelope
   distributeTokens();           // devolve as marcas atualizadas
}
```

Figura 31 – Método *run* da classe *Indication_Feedback*.

A conexão entre a RdP e os elementos externos, como dados vindos dos dispositivos ou dados específicos da aplicação (como lista de objetos, apontadores, menus, etc) pode ser feita

```
void Indication_Feedback::preProcessingTokens() {
  // obtém uma referência ao conector que recebe o objeto selecionado
  ConnectorIn *tmp = findConnIn("objeto selecionado");
  if (tmp != NULL) {  // se houver o conector...
    // pega e desempacota o objeto que está dentro da marca
    objSel = (objGeometrico*)tmp->unPackFromToken();
  }
}
```

Figura 32 – Método *preProcessingTokens* da classe *Indication_Feedback*.

```
void Indication_Feedback::distributeTokens() {
  // obtém uma referência ao conector que envia o objeto selecionado
  ConnectorOut* tmp = findConnOut("objeto selecionado");
  if (tmp != NULL) {  // se houver o conector...
    tmp->packInToken(&objSel); // empacota o objeto em uma marca
    tmp->dispatch();  // envia a marca através do conector
  }
}
```

Figura 33 – Método *distributeTokens* da classe *Indication_Feedback*.

através do método *insertToken*, disponível na *classe PetriNet* (veja Figura 29, linha 10). Esta função-membro permite que **marcas** sejam adicionadas e atualizadas em **lugares**, tanto na criação da RdP, quanto durante sua simulação.

Após todas essas etapas, a execução da RdP pode ser iniciada através da chamada do método *run* do objeto da classe *PetriNet* (Figura 29, linha 11). Este procedimento executa os métodos *run* de todos os objetos adicionados à sua lista. Para garantir a sincronização entre a função de desenho da aplicação e a simulação da RdP, o desenvolvedor precisa evocar o método *run* da **rede** no início de cada ciclo de *rendering*.

Desta forma, a fase de implementação pode ser resumida nesta seqüência de passos:

- Derivar novas classes da classe base *Transition* para representar tarefas do AV;
- Instanciar o objeto que representa toda a RdP;
- Instanciar os objetos correspondentes aos **lugares**, **transições** e **marcas**;
- Adicionar estes objetos à **rede**;

- Adicionar conectores para **lugares** e **transições**;
- Realizar a ligação entre **lugares** e **transições**, através dos conectores;
- Definir o valor das **marcas** iniciais;
- Definir pontos onde a aplicação atualizará a RdP, através da inserção/atualização das **marcas**;
- Executar a RdP.

5.6 Testes do Modelo

Após modelada e implementada a aplicação de Quebra-Cabeça Virtual utilizando a metodologia proposta, alguns testes foram realizados para observar o comportamento do modelo e da aplicação em situações críticas como dados inválidos, errados ou redundantes.

O modelo previamente apresentado descreve situações válidas que podem ocorrer durante o processo interativo. Com dados válidos, a aplicação foi testada por diferentes usuários, com duas configurações de dispositivos. A primeira caracterizou-se por representar um AV não-imersivo onde a interação era realizada através de dispositivos convencionais (*mouse*, teclado e monitor). Já a segunda representou um AV imersivo, com o uso de um óculos de RV (*I-glasses!*) e um dispositivo rastreador de posição (*Polhemus Isotrack II*), fixado a um *mouse*. Em ambas, a RdP executou apropriadamente.

No entanto, algumas situações inválidas podem ocorrer impedindo a execução da rede e, conseqüentemente, bloqueando a interação no AV. Para tanto, é importante verificar se o modelo comporta-se adequadamente nestes casos, não atuando de maneira inesperada.

O primeiro teste procurou simular um AV vazio, sem objetos para seleção. Desta forma, quando o usuário pressionasse o botão de seleção para indicar um objeto, nada poderia acontecer. Do ponto de vista do modelo, esta situação pode ser representada pela exclusão da **marca** *Lista de Objetos* da RdP (veja Figura 34). Com esta configuração, a RdP continuou executando normalmente, porém, seu novo comportamento impediu que as transições *Subtarefa de Indicação* e *Subtarefa de Feedback de Indicação* fossem disparadas, como era esperado.

Em uma segunda simulação, optou-se pela exclusão do objeto apontador, representado na RdP pela ausência da **marca** *Mão Virtual* (veja Figura 35). Assim como na situação anterior, a rede continuou executando normalmente, porém impedindo o usuário de realizar qualquer tarefa no AV, novamente conforme esperado.

No terceiro teste, inseriu-se **marcas** que não pertenciam ao conjunto de **marcas** esperado. O objetivo era verificar se estas influenciariam ou não no comportamento da RdP. Para tanto, dados vindos do rastreador, referentes à sua orientação, foram introduzidas no **lugar** *Estado de*

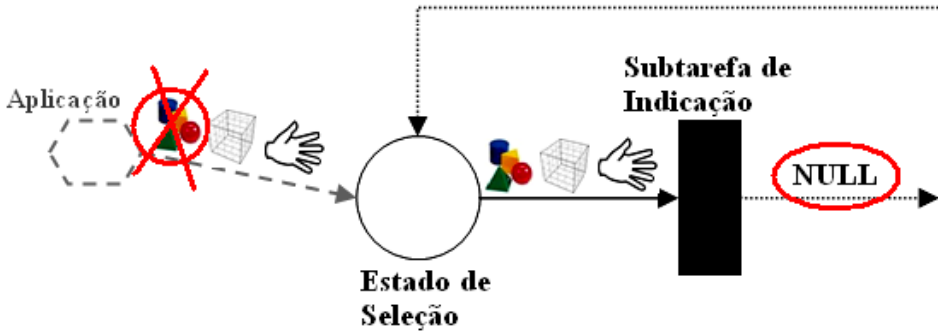


Figura 34 – O modelo sem a **marca** *Lista de Objetos*.

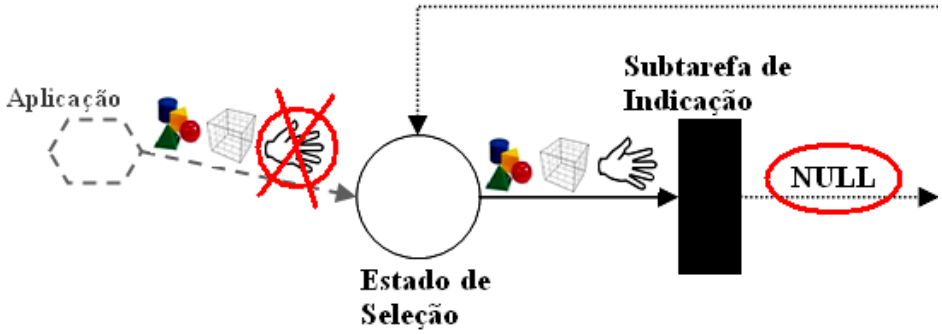


Figura 35 – O modelo sem a **marca** *Mão Virtual*.

Manipulação (veja Figura 36). Conforme esperado, estes dados foram descartados quando a **transição** *Subtarefa de Posicionamento* era disparada.

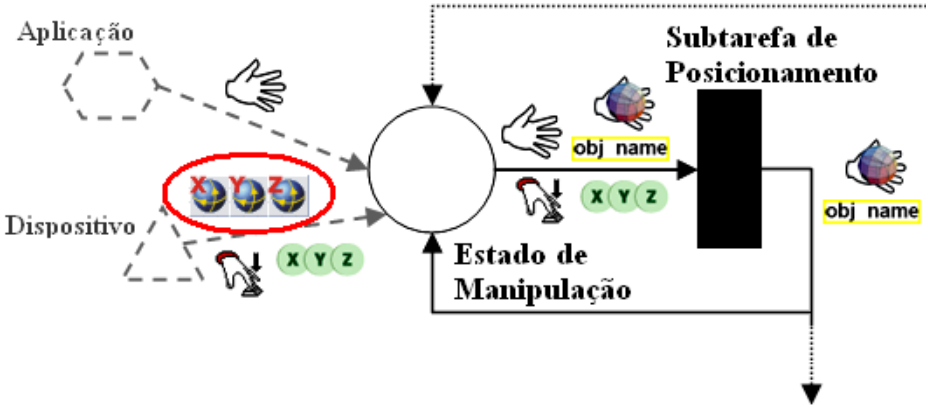


Figura 36 – O modelo com a **marca** *Orientação 3D*.

O objetivo do último teste foi simular uma situação onde não houvesse comunicação entre o AV e a RdP (veja Figura 37). Como conseqüência, o conteúdo das **marcas** da RdP era vazio, o que impedia a execução da aplicação pois nenhuma **transição** era habilitada.

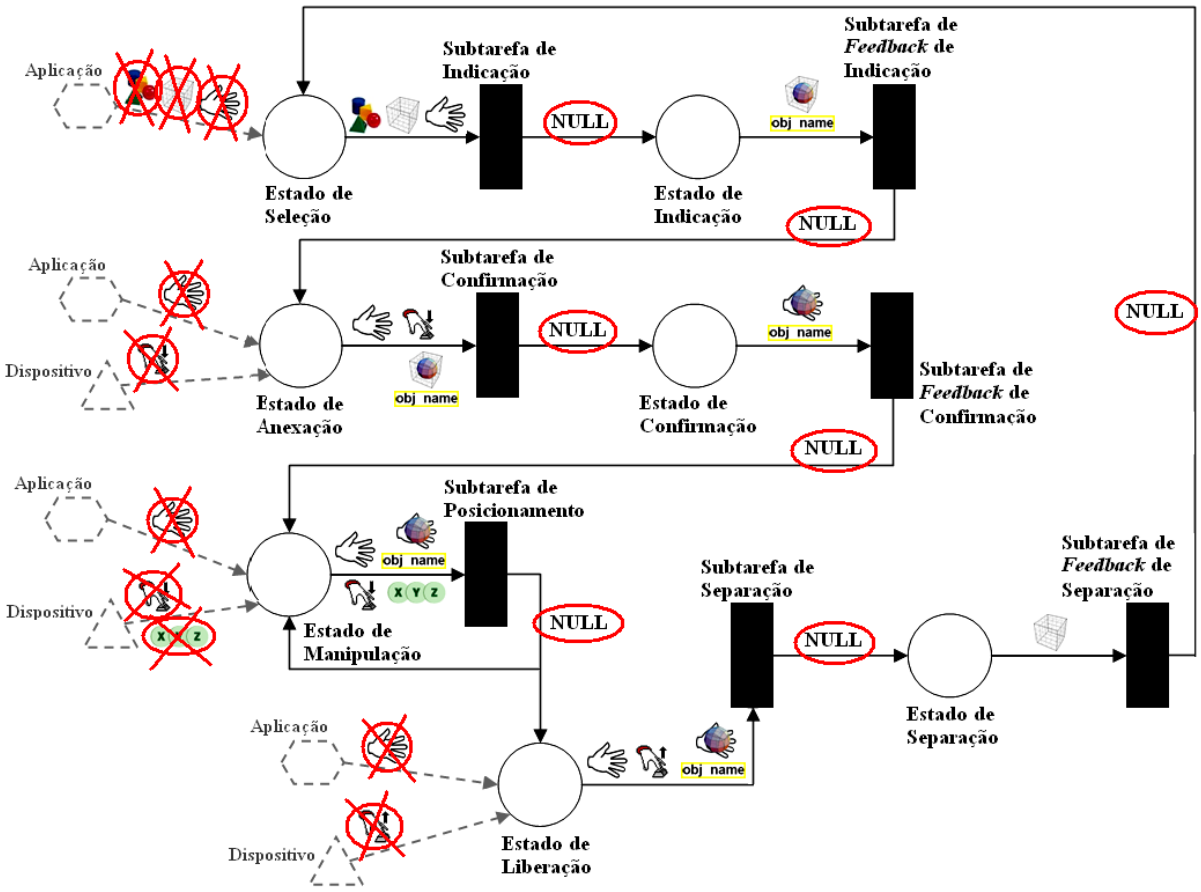


Figura 37 – O modelo sem receber informações de dispositivos ou da aplicação.

5.7 Modelagem Hierárquica

A modelagem hierárquica de uma aplicação usando RdP permite descrever o sistema em diferentes níveis de abstração, simplificando a representação e oferecendo diferentes visões do mesmo sistema.

Além disso, a existência de níveis de hierarquia pode ser útil para tornar mais claro o funcionamento do modelo como, por exemplo, em situações onde um conjunto complexo de operações necessite de uma representação simplicada, em um único módulo como uma técnica de interação, por exemplo.

A título de exemplo, imagine-se que seja interessante agrupar em uma única **transição** todo o processo de seleção. Desta forma, as **transições** *Subtarefa de Indicação*, *Subtarefa de Feedback de Indicação*, *Subtarefa de Confirmação* e *Subtarefa de Feedback de Confirmação*, e os **lugares** *Estado de Indicação*, *Estado de Confirmação* e *Estado de Anexação* podem ser agrupados como uma única entidade, denominada *Tarefa de Seleção*. As **transições** e **lugares** hachurados da Figura 38 destacam o conjunto a ser agrupado.

Já a Figura 39 mostra o novo modelo como uma RdP-H. Neste caso, a interpretação do modelo continua praticamente a mesma, pois a **transição** *Tarefa de Seleção* recebe as **marcas**

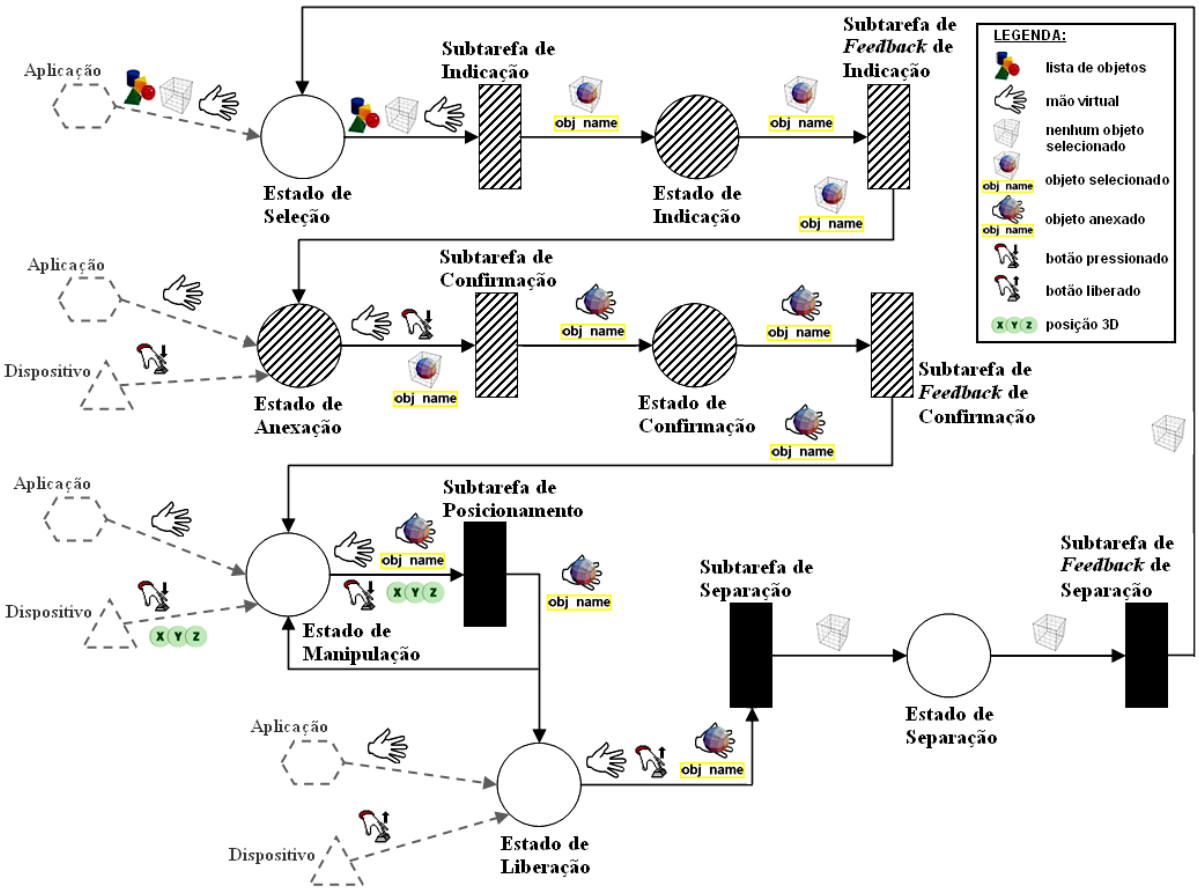


Figura 38 – RdP da Figura 27 apresentando os nodos a serem integrados.

vindas do *Estado de Seleção* e repassa a **marca** *Objeto Selecionado* (indicado e confirmado) para o *Estado de Manipulação*.

Para representar um subconjunto de lugares e transições (sub-redes), uma nova entidade precisa ser definida. Nesta metodologia, estas entidades são chamadas de **páginas** e podem ser instanciadas a partir das classes *PlacePage* e *TransitionPage*. A primeira abstrai uma rede que inicia e termina por **lugares**, enquanto a segunda encapsula uma rede que inicia e termina por **transições**. Note-se que para manter a consistência da RdP é necessário que a **página** inicie e termine por elementos da mesma classe.

Para este exemplo, optou-se pela criação de um objeto do tipo *TransitionPage* para representar a *Tarefa de Seleção*. Posteriormente, este objeto foi adicionado a RdP, substituindo os elementos que agora fazem parte de sua lista.

As conexões entre os elementos encapsulados por esta **página** permanecem inalteradas, com exceção das **transições** que localizam-se nos extremos da sub-rede e dos **lugares** que recebem informações vindas da aplicação ou dos dispositivos. Isto ocorre pois os "antigos" conectores, a quem estes elementos estavam conectados, precisam agora ficar diretamente ligados aos conectores da **página**.

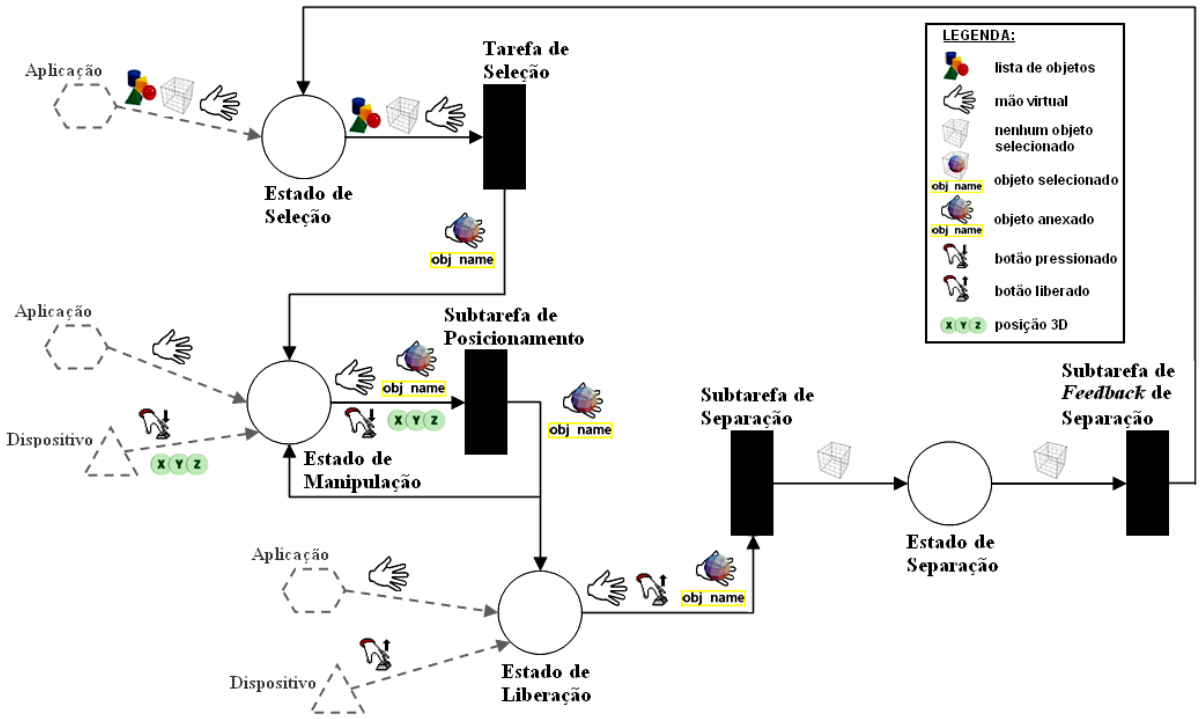


Figura 39 – RdP com a Tarefa de Seleção hierarquizada, abstraindo detalhes do modelo representado pela Figura 38.

Tome-se como exemplos as Figuras 40 e 41 (versões simplificadas do modelo da Figura 38). Na Figura 40, o conector *objs* do **lugar** *Estado de Seleção* era ligado ao conector *objs* da **transição** *Subtarefa de Indicação*, assim como o conector *objSel* da **transição** *Subtarefa de Feedback de Confirmação* era ligado ao conector *objSel* do **lugar** *Estado de Manipulação*. Agora, como mostra a Figura 41, com a modelagem hierárquica o conector *objs* do **lugar** *Estado de Seleção* passa a ligar-se ao conector *C1* da **página** *Tarefa de Seleção*, enquanto o conector *C2* da **página** *Tarefa de Seleção* liga-se ao conector *objSel* do **lugar** *Estado de Manipulação*.

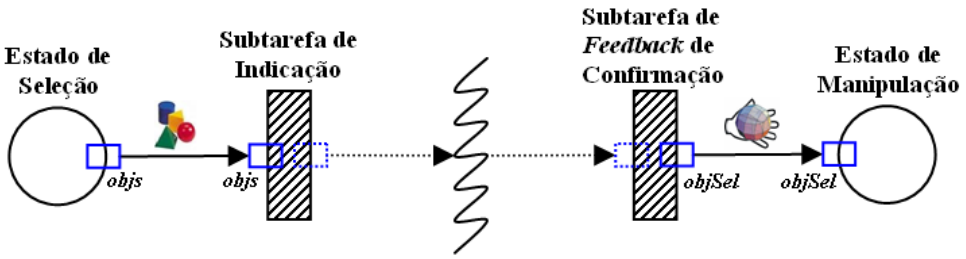


Figura 40 – Conexão entre os elementos da RdP, com base na modelagem padrão.

Para estabelecer a comunicação entre as hierarquias da RdP (setas curvas da Figura 41), as classes *PlacePage* e *TransitionPage* oferecem dois métodos responsáveis por "revelar" (expor) à rede os conectores da **página**. O método *exposeIn* revela para um elemento da sub-rede o valor depositado em um conector de entrada da **página**, enquanto que o método *exposeOut* revela para a **página** um valor depositado em um conector de saída de um elemento da sub-rede.

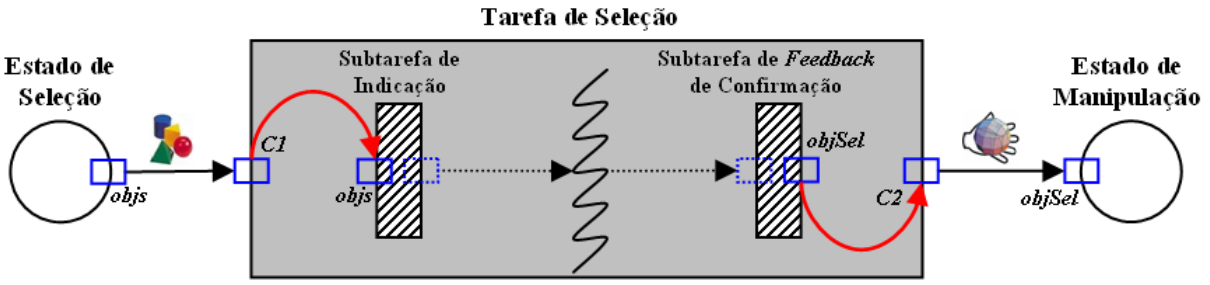


Figura 41 – Conexão entre os elementos da RdP, com base na modelagem hierárquica.

A Figura 42 apresenta um exemplo simples de geração de código do modelo hierárquico. Neste exemplo, cria-se uma **página** (linha 01) onde elementos da RdP são à ela adicionados (linhas 02, 03 e 04). A seguir, definem-se os conectores de entrada e saída desta **página** (linhas 05 e 06), bem como estabelecem-se as conexões entre ela e os elementos da RdP de nível hierárquico superior (linhas 07 e 08). Após estes procedimentos, estabelece-se a comunicação entre as hierarquias utilizando os métodos *exposeIn* e *exposeOut* (linhas 09 e 10).

Na linha 09 o conector de entrada *C1* da **página** *Tarefa de Seleção* (*tPage*) expõe seu conteúdo ao conector de entrada *objs* da **transição** *Subtarefa de Indicação* (*tInd*), enquanto que na linha 10 o conector de saída *objSel* da **transição** *Subtarefa de Feedback de Confirmação* (*tFConf*) expõe seu conteúdo ao conector de saída *C2* da **página** *Tarefa de Seleção* (*tPage*).

Linha	Código
1	`TransitionPage *tPage = new TransitionPage();`
2	`tPage->addElement("Subtarefa de Indicação", tInd);`
3	`tPage->addElement("Lugar simplificado", pTemp);`
4	`tPage->addElement("Subtarefa de Feedback de Confirmação", tFConf);`
5	`tPage->addConnIn("C1", "vector");`
6	`tPage->addConnOut("C2", "objGeometrico");`
7	`pn->connectPN(pSel, "obj", tPage, "C1");`
8	`pn->connectPN(tPage, "C2", pMan, "objSel")`
9	`tPage->exposeIn("C1", tInd, "objs");`
10	`tPage->exposeOut(tFConf, "objSel", "C2");`

Figura 42 – Exemplo de código para implementar a comunicação entre as hierarquias da RdP.

Como forma de testar a aplicação com o uso desta RdP-H, os mesmos testes vistos na Seção 5.6 foram empregados. Conforme esperado, os mesmos resultados foram obtidos, comprovando a eficácia da nova representação.

Finalizando, a Tabela 5 apresenta uma adaptação da Tabela 2, incluindo as características e recursos oferecidos pela nova metodologia (com base no estudo de caso realizado).

Tabela 5 – Características das metodologias estudadas em comparação com a nova metodologia proposta.

	Características	**Hynet**	**Flownet**	**ICO**	**InTML**	**Unit**	**CHASM**	**IPN-IML**	**DWARF UIC**	**Nova Metodologia**
1	Formalismo	RdP	RdP	RdP	—	—	MEF	RdP	RdP	RdP e Taxonomia de Bowman
2	Editor de Modelos	Não	Sim	Sim	Não	Não	Não	Não	Sim	Em projeto
3	Modelos Animados	Não	Não	Sim	Não	Não	Não	Não	Sim	Em projeto
4	Codificação	—	—	—	InTML	Grafos	Texto	VRML	XML	XML
5	Geração de Código	—	—	—	Java	Java	Texto	XML	Java	C++
6	Hierarquização	Sim	Sim	Não	Sim	Sim	Não	Não	Não	Sim
7	Fases do Desenvolvimento	A, P	A, P	A, P	A, P, I	A, P, I	A, P	I	A, P, I	A, P, I

6 Conclusões

Este trabalho definiu uma metodologia para especificar tarefas de interação de uma aplicação de RV, utilizando o formalismo de RdP como base no processo da modelagem de *software*. Para validá-la, um AV foi construído utilizando a metodologia. Esta aplicação foi testada, e comportou-se conforme esperado tanto em situações normais, quanto em situações especiais, como naquelas em que a aplicação não dispunha dos dados necessários para a execução do programa.

A expectativa é que esta metodologia, à medida que vá sendo empregada, permita que tarefas e técnicas de interação possam tornar-se componentes disponíveis para novos sistemas, uma vez que sejam testadas e simuladas previamente.

Outro objetivo desta metodologia é auxiliar no desenvolvimento de aplicações, aproximando a etapa de concepção e projeto de AVs à etapa de desenvolvimento, oferecendo recursos para implementação, testes e documentação do sistema. Além disso, é possível desvincular da metodologia a taxonomia de Bowman, adotando outras formas de categorização que preservem a modelagem da aplicação sob a perspectiva de tarefas do usuário.

Apesar do processo de implementação apresentado ter usufruído dos recursos de uma biblioteca gráfica específica, não existe impedimento para o uso de outras bibliotecas. Os passos da metodologia (e o pacote de classes desenvolvido) são independentes das funcionalidades gráficas e podem ser adaptadas conforme as necessidades de outras ferramentas, como, por exemplo, OpenSceneGraph [28], Crystal Space [48] e X3D [53].

A possibilidade de importar arquivos de editores gráficos como o DIA [18] e o yEd [56] está sendo analisada. Estes editores oferecem recursos para a construção de diferentes diagramas (entre os quais, RdP), além de armazenar a descrição destes modelos em formato XML. Como trabalhos futuros, sugere-se construir um editor gráfico capaz de importar estes modelos e, a partir deles, derivar o código-fonte da aplicação automaticamente.

Como a aplicação é controlada pelos estágios de simulação da RdP, seria interessante também apresentar ao projetista o processo de execução do programa em um gráfico animado, sobreposto ao grafo da RdP, e em paralelo ao uso da aplicação.

Da mesma forma, uma idéia interessante seria incorporar esta metodologia a um *framework* de RV, tornando-o uma plataforma completa de desenvolvimento. Neste *framework* de interação, recursos para análise, projeto, desenvolvimento e avaliação de protótipos de AVs poderiam estar incorporados, permitindo que falhas de projeto fossem rapidamente detectadas e o tempo de construção dos sistemas fosse reduzido.

Neste sentido, o *framework* ViRAL (*Virtual Reality Abstract Layer*) [3] está sendo analisado

para incorporar esta metodologia, uma vez que o mesmo já utiliza um conjunto extensível de componentes de *software* que facilitam o controle, a comunicação e a organização dos módulos de um sistema. Com a introdução desta metodologia, técnicas de interação também poderiam ser representadas e oferecidas ao projetista como componentes, assim como já acontece com dispositivos, avatares e AVs.

Referências

[1] R. Bastide, D. Navarre, P. Palanque, A. Schyn, and P. Dragicevic. A model-based approach for real-time embedded multimodal systems in military aircrafts. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 243–250, New York, NY, USA, 2004. ACM Press.

[2] R. Bastide, P. A. Palanque, D.-H. Le, and J. Munoz. Integrating rendering specifications into a formalism for the design of interactive systems. In P. Markopoulos and P. Johnson, editors, *DSV-IS*, volume 1, pages 171–190. Springer, 1998.

[3] T. A. Bastos, R. J. M. da Silva, A. B. Raposo, and M. Gattass. Viral: um framework para o desenvolvimento de aplicações de realidade virtual. In *SVR '04: Proceedings of the 7th Symposium on Virtual Reality*, pages 51–62, 2004.

[4] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner. Design of a component-based augmented reality framework. In *Proceedings of the International Symposium on Augmented Reality (ISAR)*, pages 45–54. IEEE Computer Society, Oct 2001.

[5] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[6] D. A. Bowman and L. F. Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 35–38, New York, NY, USA, 1997. ACM Press.

[7] D. A. Bowman and L. F. Hodges. Formalizing the design, evaluation, and application of interaction techniques for immersive virtual environments. *Journal of Visual Languages and Computing*, 10(1):37–53, 1999.

[8] D. A. Bowman, E. Kruijff, J. J. L. Jr., and I. Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley, 2005.

[9] R. Dachselt and A. Hübner. A survey and taxonomy of 3d menu techniques. In *EGVE '06: Proceedings of the 12th Eurographics Symposium on Virtual Environments*, pages 89–99, 2006.

[10] H. Deitel and P. Deitel. *C++ How to Program*. Prentice Hall, 2003.

[11] P. Figueroa, M. Green, and H. J. Hoover. Intml: a description language for vr applications. In *Web3D '02: Proceeding of the seventh international conference on 3D Web technology*, pages 53–58, New York, NY, USA, 2002. ACM Press.

[12] P. Figueroa, J. H. Hoover, and P. Boulanger. Intml concepts. Technical Report TR 04-06, Department of Computing Science, University of Alberta, may 2004.

[13] M. Fowler. *Analysis patterns: reusable objects models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[15] O. Hilliges, C. Sandor, and G. Klinker. A lightweight approach for experimenting with tangible interaction metaphors. In *Proc. of the International Workshop on Multi-user and Ubiquitous User Interfaces (MU3I)*, pages 51–52, 2004.

[16] I. Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Wokingham, UK, 1992.

[17] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer, 1997. Three Volumes.

[18] A. Larsson and et. al. DIA, a drawing program - GNOME-related project. Available on: http://www.gnome.org/projects/dia/, 2006.

[19] R. W. Lindeman. *Bimanual interaction, passive-haptic feedback, 3d widget representation, and simulated surface constraints for interaction in immersive virtual environments*. PhD thesis, Faculty of the School of Engineering and Applied Science, George Washington University, 1999. Director-James K. Hahn.

[20] P. R. M. Maciel, R. D. Lins, and P. R. F. Cunha. *Introdução às Redes de Petri e Aplicações*. Instituto de Computação, UNICAMP, 1996.

[21] A. MacWilliams, C. Sandor, M. Wagner, M. Bauer, G. Klinker, and B. Brügge. Herding sheep: Live system development for distributed augmented reality. In *ISMAR*, pages 123–132. IEEE Computer Society, 2003.

[22] K. H. Mortensen and et. al. Petri Nets World: Online Services for the International Petri Nets Community. Available on: http://www.informatik.uni-hamburg.de/TGI/PetriNets/, 2006.

[23] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989. NewsletterInfo: 33Published as Proceedings of the IEEE, volume 77, number 4.

[24] D. Navarre, P. A. Palanque, R. Bastide, A. Schyn, M. Winckler, L. P. Nedel, and C. M. D. S. Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In M. F. Costabile and F. Paternò, editors, *INTERACT*, volume 3585 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2005.

[25] D. Navarre, P. A. Palanque, R. Bastide, and O. Sy. Structuring interactive systems specifications for executability and prototypability. In *DSV-IS*, pages 97–119, 2000.

[26] M. Nowostawski. Jfern, Java-based Petri Net framework. http://sourceforge.net/projects/jfern, 2006.

[27] A. Olwal and S. Feiner. Unit: modular development of distributed interaction techniques for highly interactive user interfaces. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 131–138, New York, NY, USA, 2004. ACM Press.

[28] R. Osfield and D. Burns. Open Scene Graph. Available on: http://www.openscenegraph.org/, 2006.

[29] P. A. Palanque. Interactive Cooperative Objects. Available on: http://liihs.irit.fr/palanque/ICOs.htm/, 2006.

[30] P. A. Palanque and R. Bastide. Synergistic modelling of tasks, users and systems using formal specification techniques. *Interacting with Computers*, 9(2):129–153, 1997.

[31] D. O. Penha, H. C. Freitas, and C. A. P. S. Martins. *Modelagem de Sistemas Computacionais usando Redes de Petri: aplica cão, análise e avalia cão*. Anais da Escola Regional de Informática - ERI RJ/ES, 2004.

[32] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM No 3, 1962. Director-James K. Hahn.

[33] J. S. Pierce, B. C. Stearns, and R. Pausch. Voodoo dolls: seamless interaction at multiple scales in virtual environments. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 141–145, New York, NY, USA, 1999. ACM Press.

[34] M. S. Pinho. Interação em ambientes tridimensionais. In *WVR '03 - Proceedings of the 3rd Workshop on Virtual Reality - Tutorial text book*, 2000.

[35] M. S. Pinho. SmallVR: Uma ferramenta orientada a objetos para o desenvolvimento de aplicações de realidade virtual. In *SVR '02: Proceedings of the 5th Symposium on Virtual Reality*, pages 329–340, 2002.

[36] M. S. Pinho. SmallVR. Available on: http://www.smallvr.org/, 2006.

[37] I. Poupyrev, M. Billinghurst, S. Weghorst, and T. Ichikawa. The go-go interaction technique: non-linear mapping for direct manipulation in vr. In *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 79–80, New York, NY, USA, 1996. ACM Press.

[38] I. Poupyrev, S. Weghorst, M. Billinghurst, and T. Ichikawa. A framework and testbed for studying manipulation techniques for immersive VR. In D. Thalmann, editor, *ACM Symposium on Virtual Reality Software and Technology*, pages 21–28, New York, NY, 1997. ACM Press.

[39] A. B. Raposo. *Coordenação em Ambientes Colaborativos usando Redes de Petri*. PhD thesis, UNICAMP, Univerdade Estadual de Campinas, 2000. L. P. Magalhães and I. L. M. Ricarte.

[40] R. Rieder, F. B. Silva, A. B. Trombetta, R. A. Kopper, M. C. dos Santos, and M. S. Pinho. Uma avaliação do uso de estímulos táteis em um ambiente virtual. In *SVR '06: Proceedings of the 8th Symposium on Virtual Reality*, pages 135–146, 2006.

[41] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, New York, NY, 1991.

[42] S. Smith and D. Duke. The hybrid world of virtual environments. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 297–308. The Eurographics Association and Blackwell Publishers, 1999.

[43] S. Smith and D. Duke. Virtual environments as hybrid systems. In *Eurographics UK 17th Annual Conference (EG-UK'99)*, Cambridge, UK, 1999.

[44] R. Stoakley, M. J. Conway, and R. Pausch. Virtual reality on a wim: interactive worlds in miniature. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 265–272, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[45] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[46] Sun. Java 3D API. Available on: http://java.sun.com/products/java-media/3D/, 2006.

[47] Trolltech. Qt Reference Documentation - Open Source Edition. Signals and Slots. Available on: http://doc.trolltech.com/4.1/signalsandslots.html, 2006.

[48] J. Tyberghein and et. al. Crystal Space 3D. Available on: http://www.crystalspace3d.org/, 2006.

[49] R. Wieting. Hybrid high-level nets. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 848–855, 1996.

[50] J. S. Willans and M. D. Harrison. Prototyping pre-implementation designs of virtual environment behaviour. In *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, pages 91–108, London, UK, 2001. Springer-Verlag.

[51] J. S. Willans and M. D. Harrison. A toolset supported approach for designing and testing virtual environment interaction techniques. *International Journal of Human-Computer Studies*, 55(2):145–165, 2001.

[52] C. A. Wingrave and D. A. Bowman. Chasm: Bridging description and implementation of 3d interfaces. In *Proceedings of the Workshop on New Directions in 3D User Interfaces*, pages 85–88, 2005.

[53] X3D. Web3D Consortium - Royalty Free, Open Standards for Real-Time 3D Communication. Available on: http://www.web3d.org/, 2006.

[54] J. Ying. An approach to Petri net based formal modeling of user interactions from x3d content. In *Web3D '06: Proceedings of the eleventh international conference on 3D web technology*, pages 153–157, New York, NY, USA, 2006. ACM Press.

[55] J. Ying and D. Gracanin. Petri net model for subjective views in collaborative virtual environments. In A. Butz, A. Krüger, and P. Olivier, editors, *Smart Graphics*, volume 3031 of *Lecture Notes in Computer Science*, pages 128–134. Springer, 2004.

[56] yWorks the diagramming company. yEd - Java Graph Editor. Available on: http://www.yworks.com/products/yed/, 2006.

A Artigo submetido para avaliação do IEEE Symposium on 3D User Interfaces 2007

A Methodology to Specify 3D Interaction using Petri Nets

Rafael Rieder*
Faculty of Computer Science
PUCRS - Brazil

Alberto B. Raposo†
Tecgraf - CG Technology Group
PUC-Rio - Brazil

Marcio S. Pinho‡
Faculty of Computer Science
PUCRS - Brazil

ABSTRACT

This work presents a methodology to model and to implement interaction tasks in virtual environments combining three orthogonal components. We use Bowman's taxonomy to divide interaction tasks in elementary components, Petri Nets formalism to describe the interaction behavior and sequence and Object Oriented Programming concepts to organize the implementation. The general goals for using these tools are to obtain a more well-structured development process, to allow the software specification to be understood by both the user and designers and to derive the software implementation in a smooth way from the specification. Besides that, we seek to provide a tool to facilitate the reuse of developed codes for elementary interaction tasks.

Keywords: interaction tasks, Petri nets, specification.

Index Terms: H.5.2 [Information Interfaces and Presentation]: User Interfaces—Theory and methods; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality I.6.5 [Simulation and Modeling]: Model Development—Modeling methodologies

1 INTRODUCTION

The Virtual Reality (VR) development application process still uses a very *ad-hoc* modeling and implementation process, with very low level standardization and almost no formalism. This is particularly noticeable in efforts devoted to scientific applications, leading, in most cases, to code rewriting and hindering application analysis before its implementation.

In order to easily understand a computation application it is very helpful to use some kind of formal description tool like Petri Nets (PN), Unified Modeling Language (UML) and Finite State Machines (FSM) that can describe the system behavior and components. These tools allow to better understand and evaluate each phase of the system operation, and moreover, facilitate the automatic generation of the application code from graphical representations and makes the application view and modeling from different abstraction levels easier.

Smith [16] points out that the lack of formal descriptions during the development process of virtual environments (VEs) inhibits the identification of similarities among different interaction techniques (ITs), leading to the "reinvention" of existing techniques. Furthermore, according to Navarre [8], informal descriptions facilitate the rise of ambiguities during the implementation process.

Various formalisms have been presented for ITs modeling, like Hynet [20], ICO [12] and Flownet [21], based on Petri Nets. The use of such tools helps, for example, to detect system failures during the early stages of the design process.

*e-mail: rieder@inf.pucrs.br
†e-mail: abraposo@tecgraf.puc-rio.br
‡e-mail:pinho@pucrs.br

Beyond formal specification tools, some researches have sort to develop taxonomies able to document and specify VEs in an abstraction level closer to the user's conception instead of the programmer's view of the application.

According to Lindeman [6] the VR field is a technology that is still in a definition stage and for this reason needs to be classified, categorized and better organized to be easily understood. Bowman [1] argues that the existence of such classifications can also be used to better organize the VR appliciation design process.

Those works that present taxonomies seek to identify the interaction process phases [1], to classify ITs [2] [6] [14] and to organize the system control [3]. These approaches split the systems in smaller parts, identifying behaviors and allowing to encapsulate them into components that are able to execute some relevant functionality. This approach allows the reuse of these components in other projects and also allows the combination of them to build a new IT, for example.

The use of formalisms and taxonomies both aim to reduce the time spent during the project for design and implementation of VEs. Therefore, an integration of these approaches can allow for system specification according to the user's level of expertise, besides allowing for the detailing of each phase of the software development process.

Based on these previous works, our project describes a methodology able to model and to implement software components that represent the interaction process phases.

The combination of these elements allows for the description of an interaction task, in which the sequence of the interaction process is controlled by a PN.

Our methodology integrates three modeling approaches: PN formalism, technique-decomposition taxonomy created by Bowman [1] and Object Oriented Programming Concepts. The PNs are used to graphically represent the VE behavior, based upon the interaction process phases of Bowman's taxonomy. The adoption of these approaches creates a model that can be easily coded using a set of C++ classes. A PN simulator is used to control the program execution flow.

The choice for the PN to specify the task in VEs emerges naturally when we start using the Bowman's methodology because the interaction tasks can be easily understood as transitions, while the states reached by the application can be understood as places in the PN. From this definition it is easy to define independent components to represent the system functionalities. The logical separation in components is important, for example, to develop IT frameworks, or to facilitate the automatic code generation from a tested and validated model.

This document is organized as follows: first we present a literature review in Section 2. Section 3 presents the developed methodology. In Section 4 we present a case study applying the methodology, and we show the necessary steps from the application modeling to the code generation for a VE. Section 5 presents the validation test for the specification. Section 6 shows the possibility of hierarchical modeling using our methodology. Section 7 closes the paper with the potentialities of our approach and the goals we want to achieve with future works.

2 RELATED WORKS

Different mechanisms have been proposed by the VR community to describe and implement ITs, seeking to understand the dynamic behavior of the applications and allowing the standardization of important functions.

Hynet [20] is a specification methodology for ITs that integrates three modeling approaches. The High-level PNs represent the formal base for the specification, defining the application semantics and allowing a graphical representation for the application events (the discrete part of the application). The Differential Algebric Equations handle the continuous behavior and Object Oriented Concepts allow to enhance the methodology expressiveness generating concise and compact models.

Based upon HyNet, the Flownet methodology [21] was developed for describing dynamic behavior in VEs and presents as a differential, a graphical notation that allows for the specification of both the discrete and continuous behavior of the application.

The Interactive Cooperative Objects (ICO) is a formal notation devoted to the specification of interactive systems [12]. It borrows concepts from the object-oriented programming to describe the structural or static aspects of systems, and uses high-level Petri nets to describe their dynamic aspects. The specification created using ICO can be simulated which gives the possibility to prototype and test quickly an application before it is fully implemented [11].

We can also view the development process of VEs under the perspective of the ITs used, with the aim of classifying them in order to better understand their components, and therefore the possibilities of software reuse in new applications.

Lindeman [6], for example, demonstrates a taxonomy that divides ITs according to the type of manipulation technique (direct or indirect), the system actions (discrete or continuous) and the degrees of freedom controlled by the IT. This approach helps to identify the parameters involved in each IT, facilitating the building of new forms of interaction.

Bowman [2] presents two complementary taxonomies to classify ITs. The first is a metaphor-based taxonomy that seeks to facilitate the user's understanding of the ITs used in a particular VE. The idea is to take some real action or situation as a frame of reference for the user to understand how to interact in the VE. The second classification, based on task decomposition, aims to perform a detailed analyses of the interaction process. According to the authors, the separation of tasks in simpler components allows each of them to be analyzed and tested in an independent way as a tool for evaluating the usability and effectiveness of an IT in a particular context or VE.

VR Frameworks also try to separate functions in components, allowing for the abstraction of the complexities of some system actions, and furthermore, the reuse of these software components in many different projects.

Figueroa [4] [6] proposes an architecture for IT development based on *pipes* and *filters*, where information sources, like physical devices generate a flow of data that are propagated through interconnected filters. This work presents the InTML markup language, based on X3D [23], to act as front-end for VR development libraries. Using this methodology, ITs can be built and used as external components, independent from the application. This approach allows to integrate existing ITs and create new ones, facilitates software reuse and decreases the VE complexity.

Similar to InTML, the Unit framework [9], also inserts an abstraction layer between the applications and its devices, and inserts application *units* into a data flow. Each unit has many different properties and can be interconnected to many other units. Moreover, the framework also allows the replacement of ITs in run-time, as they have been specified during the project phase.

Wingrave [22] proposes an architecture based on hierarchical state machines that is responsible for the communication between the designer and the programmer and between the programmer and the three dimensional interface, easing the code management and its reuse.

Using a different perspective from the previous presented works, Ying [25] aims to understand the interaction process of existing VR applications. Analyzing an existing code, the information related to the user interaction is extracted and organized in an XML file that serves as a base for building a PN model representing the target application. Simulating the PN one can "view" the interaction process through the PN behavior, while the user is interacting with the VR application.

From the literature review, it is possible to perceive that the approaches have specific advantages and goals, but, in general, do not address the entire computer application development cycle. Towards this goal, this project creates a methodology for hierarchical development of VR interaction process, from the design stage, based upon an interaction taxonomy, to the implementation phase, relying on the object oriented programming paradigm.

3 METHODOLOGY BASIS

Our methodology uses PNs because they are a very well know formalism and have many variations [24]. According to Murata [7] a PN is a graphical and mathematical modeling tool to specify and analyze dynamic and concurrent systems, amongst which, we can include VR applications.

Moreover, the power of expression of its formalism, combined with the existing tools for simulation and analysis, allows PNs to be used to preview and test applications before really implementing them.

The graphical representation adopted here is based on *Colored Petri Nets* (CPNs), because we need an easy and short way, in the modeling process, to differentiate the various types of data that are manipulated in a VR application. In fact, for CPNs, the data types can be represented by different colors, different patterns or even different icons. For simplication, this work refers to CPNs simply by the expression Petri Nets (PN).

The methodology presented in this paper aims to approximate the user's conception of the application from the designer and developer's point of view, modeling the application under the perspective of tasks which the user has or wishes to perform inside the VE.

These tasks can be decomposed in elementary-tasks that can be easily identified in most VR applications. These elementary-tasks, according to Bowman [1] split the interaction process into three phases: **selection**, **manipulation** and **release**. Our work adapts these taxonomy dividing the selection phase into **selection** and **attachment**. The former represents the indication of the object which the user wishes to manipulate. The latter deals with the confirmation of this selection. Both provide feedback to the user in order to confirm their execution. The **manipulation** (positioning and orientation) and **release** tasks remain unmodified, following Bowman's original conception.

Based on these definitions our methodology defines that each element used in the PNs (**Places**, **Transitions**, **Arcs** and **Tokens**) assumes a specific role or function during the interaction process in a VR application.

Places define the **current application state**. They have communication channels to receive and to transmit the necessary information for the PN execution. They do not produce any data.

In order to have, for example, the selection task accessible to the user, it is necessary for the **place** which represents the selection state to have at least the pointer and the objects that can be selected, and to ensure the user is not already manipulating an object. Figure 1 represents this situation using one **place** and one **transition**.

Transitions are the PN elements that **perform actions**, modifying the application behavior. They represent the tasks in the interac-

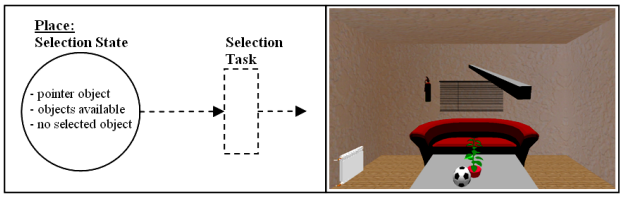


Figure 1: The *Selection state place* holds the necessary information that makes the *Selection task* accessible to the user.

tion process. Like the **places**, the **transitions** have communication channels to receive and transmit the necessary information for the PN execution, but they can also produce data and insert them into the network.

An example of this behavior can be the action performed at the exact moment the user confirms the selection of an object. A **transition** responsible for this task is fired, executing the operation that attaches the object to the selection pointer. Figure 2 models the *Attachment task transition*. Once fired, it establishes the *Manipulation state*.

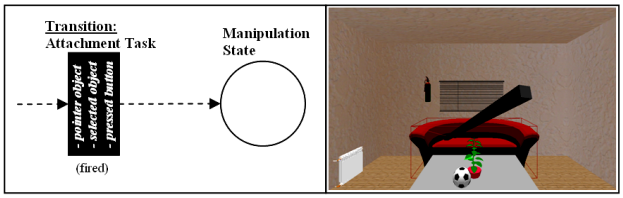


Figure 2: **Transition** that attaches the selected object to the pointer. Afterwards the application passes to the *Manipulation state*.

Arcs define the execution sequence of the PN. They are responsible for carrying data between **places** and **transitions** (and vice versa), setting pre and post conditions for **transition** firing or for establishing a state. Consequently, the **arcs** define the order of execution among tasks in the VE.

In order to manipulate an object, it is necessary to execute the selection and attachment tasks beforehand, because they provide the information about the object the user has chosen. Figure 3 shows this sequence, labeling the **arcs** with the necessary data for each task.

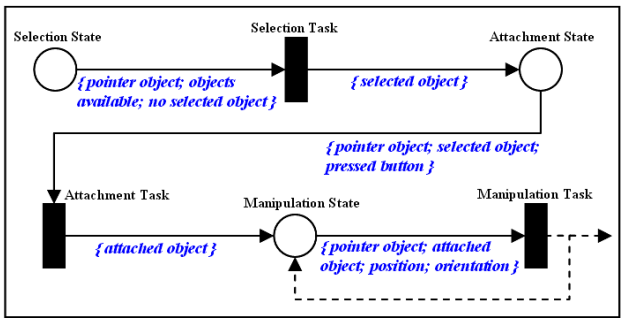


Figure 3: **Arcs** between **places** and **transitions** defining the order and the resources for each phase of the interaction process.

Tokens are the resources available for executing the application, as well as the PN. Application data like geometric objects, menus, positioning data, button clicks, and so on, are examples of possible **tokens**. These kinds of resources can come from the application or from devices like the mouse, keyboard or tracker. They can be stored inside the PN **places** and their values can be updated when they go through **transitions**. In our methodology, one **token** can be distinguished from another by the data type it encapsulates. Figure 4 shows a PN with **tokens** represented by icons.

Because the described model is based on PNs, we need to set up initial **tokens** to be able to run the simulation. From the application point of view these **tokens** represent the initial system configuration. The initial state of the devices and the list of geometric objects used in the VE can be considered as initial **tokens**. Following the PN rules we will always need **places** to store these initial data.

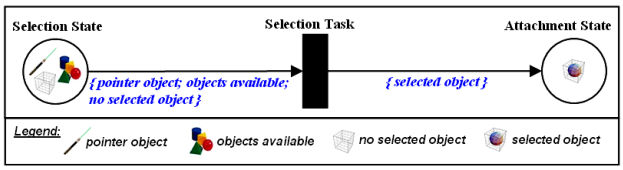


Figure 4: Objects and their representation as PN **tokens**.

It is important to point out that a **place** or a **transition** can never be directly connected to another node of the same type.

4 Applying the Methodology

4.1 Platform for Methodology Testing

In order to illustrate the use of our methodology in the specification process of a VE, we built a virtual puzzle application [15], in which the user's primary goal is to arrange some "blocks" inside a shelf. Figure 5 presents the main view of the VE. Initially, the blocks appear scrambled on the right side of the user's field of view and the shelf on his left. There is no proper order for the user to arrange the blocks. He/she uses the virtual hand technique for interacting with the blocks.

This application was built using C++, OpenGL, GLUT and the SmallVR toolkit [13], that facilitates the development of VR applications, abstracting many implementation aspects like device control and scene graph management, while maintaining the GLUT structure for the program.

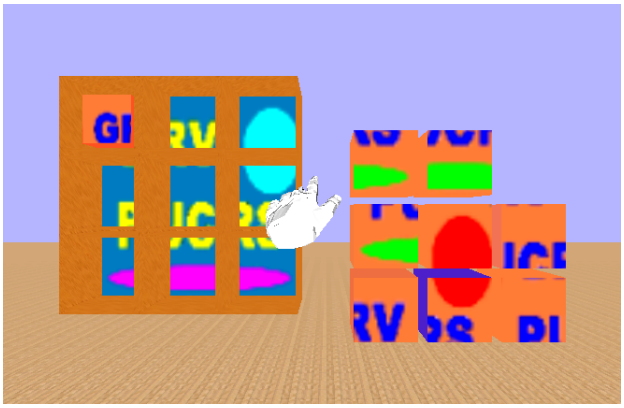


Figure 5: Virtual Puzzle initial view.

The designer needs to follow three steps to apply the methodology:

- identify the VE tasks, according to Bowman's taxonomy, as well as the main states reached by the application after executing each task;

- define a PN with the tasks and states identified in the previous step;
- implement the model, using a set of classes specially developed to build the PN and to control its execution.

4.2 Identifying Interaction Phases

The application starts in the *Selection state* (see Figure 6, on the left) in which the user can move the pointer (virtual hand) searching for an object to select. From this point the *Selection task* tests if there is a collision between the pointer and a virtual object. If there is, the *Attachment state* is established.

At this point if the user presses and holds the selection button, the *Attachment task* is fired attaching the selected object to the pointer and establishing the *Manipulation state*.

Once this state is established the PN fires the *Manipulation task* which allows the user to relocate the object using the virtual hand. In order to simplify the model, we omitted some feedback normally provided for the user.

If the user releases the selection button, it makes the *Release state* go to the enabled state and this fires the *Release task* separating the pointer from the previous selected object.

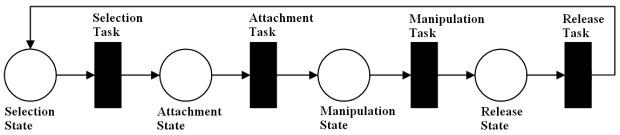


Figure 6: A high-level PN for the application modeling.

4.3 Building the PN Model

After identifying the application tasks in a high level of abstraction, we perform a task subdivision process splitting them into smaller parts (see Table 1), based upon the operations each of them has to execute. Figure 7 shows the new PN configuration.

Table 1: High-level tasks detailing.

High-Level Tasks	Basic Operations
Selection Task	- Indication Subtask - Indication Feedback Subtask
Attachment Task	- Confirmation Subtask - Confirmation Feedback Subtask
Manipulation Task	- Positioning Subtask
Release Task	- Detachment Subtask - Detachment Feedback Subtask

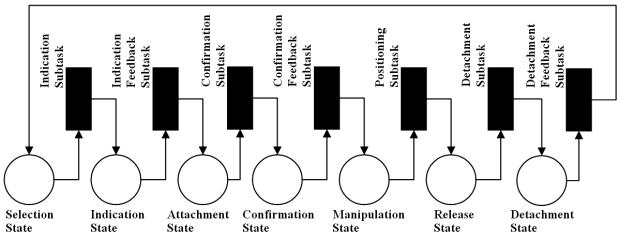


Figure 7: PN model detailing the interaction process of the Virtual Puzzle application.

Following the methodology, we should start identifying the necessary resources (data) for each interaction phase. In the PN they will be represented as **tokens**. For this the PN **arcs** should be labeled with the **tokens**, in this case represented by icons.

The **places** *Selection State*, *Attachment State*, *Manipulation State* and *Release State* need to be constantly updated with information about the devices and control variables from the application. Therefore, **tokens** with these data must be inserted into them, as can be seen in Figure 8, that presents the complete PN model for the application. In this picture the devices are represented by triangles, while the application is represented by a hexagon. These shapes are merely illustrative and serve only to help the understanding of the network behavior.

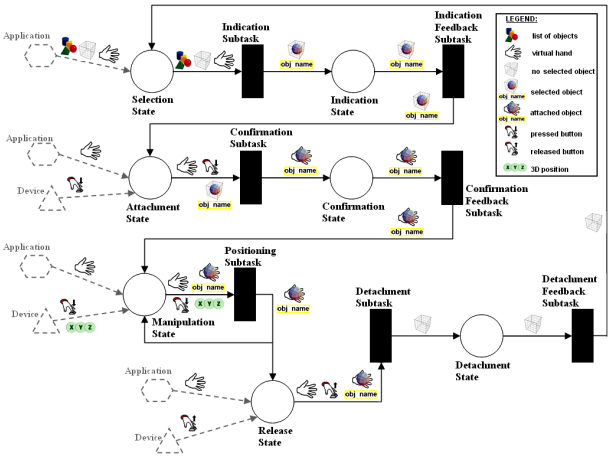


Figure 8: PN model and data resources for the interaction process in the Virtual Puzzle application.

The pace of the PN simulation is controlled by the application, that tests each **transition** every time the user's view needs to be modified. In other words, all the **transitions** are checked on every rendering cycle and fired or not, depending on the existence of the pre-conditions (the necessary **tokens**).

A complete cycle of the PN execution can be interpreted as follows: initially, the application and the devices send **tokens** to the *Selection*, *Attachment*, *Manipulation* and *Release states*. As the *Indication Subtask transition* receives all the necessary **tokens** from the *Selection state*, it is fired and can unpack its data (*list of objects*, *pointer*, *control variable* which indicates there is no object being manipulated) and test if there is a collision between the user's pointer and the one of the objects inside the VE. If there is a collision, the **transition** creates a new **token** to represent this object and passes it to the *Indication state*. Once it is established, this state fires the *Indication Feedback Subtask transition*, responsible for producing highlighting for the object in order to differentiate it from the other objects in the VE.

Immediately the *Attachment state* receives the **token** that represents the selected object. This state already has the pointer **token** and maybe the information that the button has been pressed by the user. When all these **tokens** are present at the state, the *Confirmation Subtask transition* is fired, attaching the selected object to the pointer.

The **token** that represents the selected object is then sent to the *Confirmation state* which then fires the *Confirmation Feedback Subtask transition* which itself produces a beep informing the user of the success of the attachment process. Afterwards, a **token** encapsulating the name of the selected object is sent to the *Manipulation state* which defines the start of the manipulation process. As this state always has the **tokens** that represent the pointer and the

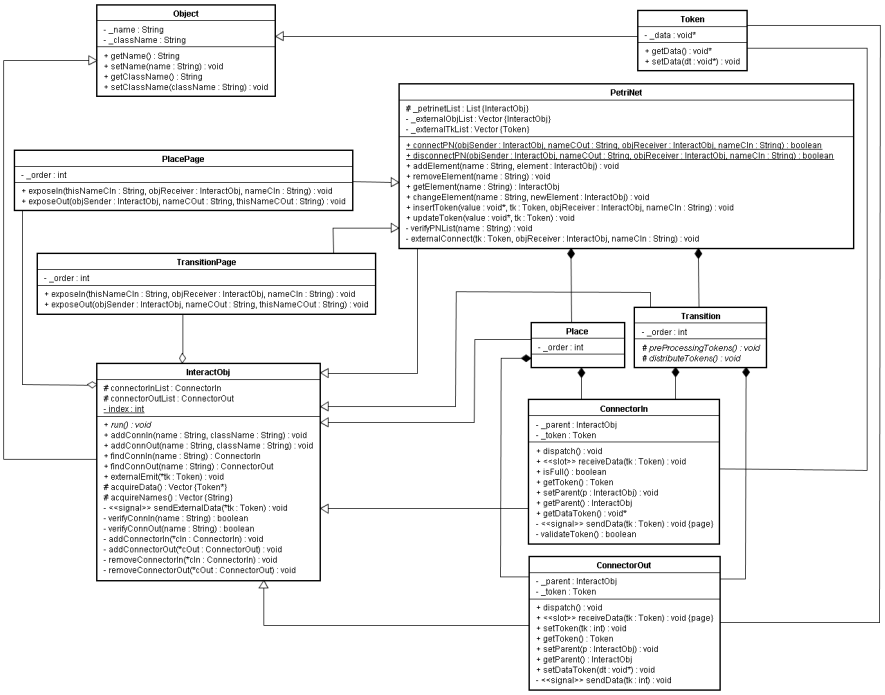


Figure 9: Classes used to implement the PN model.

data that comes from the tracker device and, providing that the button is still pressed, as soon as it receives the selected object **token** it fires the *Positioning Subtask transition*. This will update the object position based on the tracker data received from the *Manipulation state*. After this the transition generates a new **token** with the selected object name and sends it both to the *Release state* and back to the *Manipulation state*. While the selection button remains pressed, the *Positioning Subtask transition* is repeatedly fired allowing the user to put the object wherever he wants.

If the button is released, the *Manipulation state* will no longer fire *Positioning Subtask transition*. Concurrently, the *Release state* receives a **token** informing the user that he/she has just released the previous selected object and it also receives a **token** that encapsulates the object name. As the information about the virtual hand is always available, the *Detachment Subtask transition* is fired. This **transition** releases the object in its new position inside the VE. Immediately, the *Detachment state* receives a **token** that fires the *Detachment Feedback Subtask transition* which then produces a beep informing the user of the success of the detachment process. A **token** is then sent to the *Selection state*, allowing a new selection to be initiated.

4.4 Implementation Phase

Once the modeling process is concluded, the next step is to derive the implementation. In order to facilitate the code generation process, we developed a set of C++ classes, described by the UML diagram presented in Figure 9. These classes represent the PN nodes and use *signal* and *slot* mechanisms [18] as the communication tools between **places** and **transitions**.

According to Trolltech [18] *signals* are notifications (messages) from one object to another and indicate the occurrence of an event while *slots* are the answers generated by the recipient object in response to these messages. In an object orientated programming paradigm this answer is a method that should be executed. In our project these mechanisms were implemented using the *QObject class* from the Qt Toolkit [17]. Using such a tool, **places** and **transitions** can be implemented as interconnected objects, facilitating the passing of **tokens** during the PN simulation.

The classes that define **places**, **arcs** and **tokens** can be directly used to instantiate objects. Nevertheless, **transitions** must be implemented from new classes derived from the abstract class that represents a generic **transition**. This approach forces the developer to implement some methods for all the **transitions** that are essential for the model simulation.

To start the model implementation initially the designer must instantiate an object from the *PetriNet class* (see Figure 10, line 1). This object will represent the entire PN for the application and will be used to run the simulation and to store **places** and **transitions**. Moreover, it has methods to encapsulate and retrieve data in/from the **tokens**, and to connect the nodes of the PN.

After instantiating the object that represents the entire PN, we create **places**, **transitions** (see Figure 10, lines 2 and 3) and **tokens** that will traverse the network (see Figure 10, line 4). **Places** must be instantiated from the *Place class*, **transitions** from a class derived from the *Transition class* (Indication class in Figure 10), and **tokens** from the *Token class*. **Places** and **transitions** must also be added to the PN (lines 5 and 6).

As stated before in Section 3, **places** and **transitions** exchange **tokens** through communication channels. These channels are established by linking *connectors* that must be added to **places** and **transitions**. They must receive a name and a data type they can deal with (lines 7 and 8). After this, to establish the communication between two objects, the designer must explicitly connect them, defining the sender object and its connector and the receiver object and its connector (line 9).

For the **transitions** it is necessary to derive new classes from the *Transition class* to represent each **transition** defined in the model. In the example presented in Figure 7 we identify the following new classes and their role in the model:

- *Indication*: detects the collision between VE objects and

user's pointer;

- *Indication_Feedback*: highlights the indicated object;
- *Confirmation*: attaches an object to the pointer;
- *Confirmation_Feedback*: notifies the attachment;
- *Positioning*: updates the object position;
- *Detachment*: releases the object;
- *Detachment_Feedback*: notifies the releasing;

As with the filters defined by Figueroa [4], the abstract class, which gave origin to the *transition derived classes*, obliges the designer to implement three methods responsible for collecting and distributing **tokens** and for processing data inside the **transitions**.

The *PreProcessingTokens* method receives **tokens**, "opens" them and has access to the application data. For its part, the *DistributeTokens* method packs the application data inside a **token** and delivers them to the PN. Finally, the *Run* allows the designer to insert the application-specific code to do whatever is necessary for application execution.

The connection between the PN and external elements as data from devices or application-specific data (like objects, pointers, menus, etc) can be made with the method *insertToken*, available in the *Place class* (Figure 10, line 10).

After all these steps, the PN execution can be started by calling the *Run* method of the *PetriNet class* (Figure 10, line 11). In order to guarantee the synchronization between the application rendering cycles and the PN simulation, the designer needs to call the *Run* method at the beginning of every rendering cycle.

Line	Code
01	`PetriNet *pn = new PetriNet();`
02	`Place *pSel = new Place();`
03	`Indication *tInd  = new Indication();`
04	`Token *tkObjs  = new Token();`
05	`pn->addElement("Selection State", pSel);`
06	`pn->addElement("Indication Subtask", tInd);`
07	`pSel->addConnOut("objs", "vector");`
08	`tInd->addConnIn("objs", "vector");`
09	`pn->connectPN(pSel, "objs", tInd, "objs");`
10	`pn->insertToken(vecObjs, tkObjs, pSel, "objs");`
11	`pn->run();`

Figure 10: A simple example of the implementation phase.

Therefore, the implementation phase for the PN model can be summarized in the following steps:

- Derive new classes from the *Transition base class* for representing the VE tasks;
- Instantiate the PN object;
- Instantiate the objects for **Places**, **Transitions** and **Tokens**;
- Add these objects to the PN object;
- Add connectors to **Places** and **Transitions**;
- Link **Places** and **Transitions**;
- Set up the value for the initial **Tokens**;
- Define where the application will update the PN;
- Execute the PN.

5 Model Evaluation

Once we had modeled and built the puzzle application following our methodology, we performed some tests to evaluate the model behavior in some critical situations like invalid, missed or redundant data.

The model previously presented describes the valid situations which can occur during the interaction process. With valid data the application has been tested by many users and the PN has run properly.

However, some important invalid situations can happen and it is important to check if the model can deal properly with them without acting in an unexpected way.

The first test simulated an empty VE (no objects in it). In this case, even when a user presses the selection button nothing should happen. In order to model this situation we removed the **token** that encapsulates the *List of Objects* data from the PN. With this configuration the PN kept running properly but nothing happened when the user pressed the selection button, as was expected.

In the second test we removed the pointer object, excluding the *Virtual Hand token* from the PN. In the same way, the application kept running properly but the user couldn't select any object, again as was expected.

For the third test we added useless **tokens** in the *Manipulation state*, representing the tracker orientation. As was desired, the data were discarded by the **transition** when it was fired.

The aim of the last performed test was to simulate a situation where there was no communication between the applications and the VE. As a consequence, the content of the **tokens** were empty thereby hindering the application execution.

6 Hierarchical Modeling

The hierarchical modeling of an application using a PN, allows to describe the system in many different levels of abstraction, simplifying the representation and giving different views for the same system. This facilitates its comprehension by people of different levels of expertise. Moreover, the possibility of representing some model parts as a unitary module (an interaction technique, for example) can also be useful, for simplifying the model.

As a matter of example, we can have a situation where it is interesting to group, in only one **transition**, the entire Selection process. In this way, the **transitions** *Indication Subtask*, *Indication Feedback Subtask*, *Confirmation Subtask* and *Confirmation Feedback Subtask*, the **places** *Indication State*, *Confirmation State* and *Attachment State* can be grouped in only one element. The **transitions** and **places** hatched in Figure 11 show the nodes to be grouped. In Figure 12 we show the new model with a hierarchical PN. In this case the model interpretation remains almost the same, because the *Selection Task transition* receives the **tokens** from the *Selection state* and passes the *Selected Object token* to the *Manipulation state*.

In order to represent the group of **transitions** and **places** a new entity called a **subnet** needs to be defined. In our methodology these entities can be instantiated from the special classes *PlacePage* and *TransitionPage*. The former abstracts a network that start and ends with a **place**, while the second one encapsulates a network that start and ends with a **transition**.

For this example we used a *TransitionPage* to represent the *Selection Task*. Afterwards, this object was added to the PN as we did with other objects. The connections between **places** and **transitions** and the **tokens**' definitions remain the same.

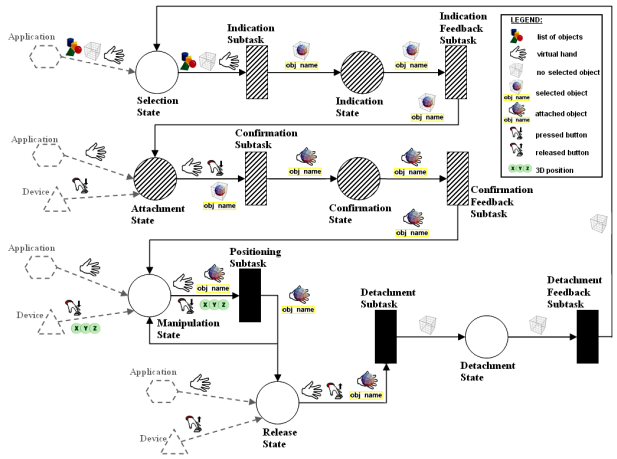


Figure 11: PN model with the candidate nodes for grouping.

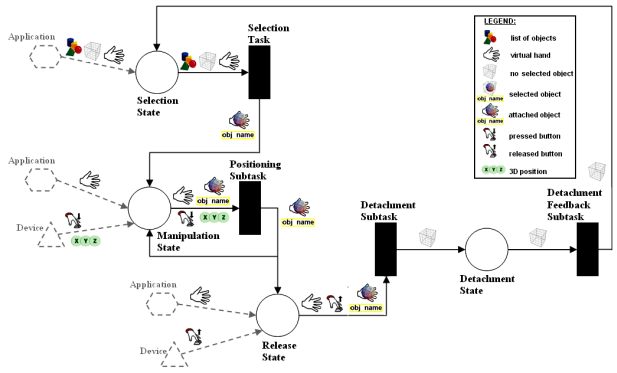


Figure 12: The PN with *Selection task*, abstracting the model represented for Figure 11.

7 Conclusions

This work presents a methodology to specify interaction tasks for VR applications using the Petri Nets formalism as a base for the software design.

Furthermore, our methodology aims to facilitate the application development from conception and design phases to the implementation, test and documentation processes.

Even though we have derived the implementation using a particular VR toolkit, there is no dependency between it and the methodology steps, thereby facilitating the replacement of such tools by any other toolkit like, for example, OpenSceneGraph [10], Crystal Space [19] and X3D [23].

In future works we plan to use a graphical editor to build the PN and derive the application source code directly from the graph description file. We are evaluating the DIA [5] and Yed [26] editors.

As we are controlling the PN simulation stages by ourselves, it would be also interesting to show the application running process in a graphical animation superimposed over the PN graph itself, in parallel with the application usage. Currently we only generate a textual output during the application execution.

Acknowledgements

This work was partially funded by Tecgraf - Computer Graphics Technology Group at PUC-Rio. We are also grateful for the fellowships granted by Dell/PUCRS Agreement and CAPES - the Brazilian Ministry of Education Agency.

References

[1] D. A. Bowman and L. F. Hodges. Formalizing the design, evaluation, and application of interaction techniques for immersive virtual environments. *Journal of Visual Languages and Computing*, 10(1):37–53, 1999.

[2] D. A. Bowman, E. Kruijff, J. J. L. Jr., and I. Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley, 2005.

[3] R. Dachselt and A. Hbner. A survey and taxonomy of 3d menu techniques. In *EGVE '06: Proceedings of the 12th Eurographics Symposium on Virtual Environments*, pages 89–99, 2006.

[4] P. Figueroa, M. Green, and H. J. Hoover. Intml: a description language for vr applications. In *Web3D '02: Proceeding of the seventh international conference on 3D Web technology*, pages 53–58, New York, NY, USA, 2002. ACM Press.

[5] A. Larsson and et. al. Dia a drawing program - gnome-related project. Available on: http://www.gnome.org/projects/dia/, 2006.

[6] R. W. Lindeman. *Bimanual interaction, passive-haptic feedback, 3d widget representation, and simulated surface constraints for interaction in immersive virtual environments*. PhD thesis, Faculty of the School of Engineering and Applied Science, George Washington University, 1999. Director-James K. Hahn.

[7] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989. NewsletterInfo: 33Published as Proceedings of the IEEE, volume 77, number 4.

[8] D. Navarre, P. A. Palanque, R. Bastide, A. Schyn, M. Winckler, L. P. Nedel, and C. M. D. S. Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In M. F. Costabile and F. Paternò, editors, *INTERACT*, volume 3585 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2005.

[9] A. Olwal and S. Feiner. Unit: modular development of distributed interaction techniques for highly interactive user interfaces. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 131–138, New York, NY, USA, 2004. ACM Press.

[10] R. Osfield and D. Burns. Open scene graph. Available on: http://www.openscenegraph.org/, 2006.

[11] P. A. Palanque. Interactive cooperative objects. Available on: http://liihs.irit.fr/palanque/ICOs.htm/, 2006.

[12] P. A. Palanque and R. Bastide. Synergistic modelling of tasks, users and systems using formal specification techniques. *Interacting with Computers*, 9(2):129–153, 1997.

[13] M. S. Pinho. Smallvr: Uma ferramenta orientada a objetos para o desenvolvimento de aplicaes de realidade virtual. In *SVR '02: Proceedings of the 5th Symposium on Virtual Reality*, pages 329–340, 2002.

[14] I. Poupyrev, S. Weghorst, M. Billinghurst, and T. Ichikawa. A framework and testbed for studying manipulation techniques for immersive VR. In D. Thalmann, editor, *ACM Symposium on Virtual Reality Software and Technology*, pages 21–28, New York, NY, 1997. ACM Press.

[15] R. Rieder, F. B. Silva, A. B. Trombetta, R. A. Kopper, M. C. dos Santos, and M. S. Pinho. Uma avalia cão do uso de estímulos táteis em um ambiente virtual. In *SVR '06: Proceedings of the 8th Symposium on Virtual Reality*, 2006.

[16] S. Smith and D. Duke. The hybrid world of virtual environments. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 297–308. The Eurographics Association and Blackwell Publishers, 1999.

[17] Trolltech. Qt. Available on: http://www.trolltech.com/products/qt, 2006.

[18] Trolltech. Qt reference documentation - open source edition. signals and slots. Available on: http://doc.trolltech.com/4.1/signalsandslots.html, 2006.

[19] J. Tyberghein and et. al. Crystal space 3d. Available on: http://www.crystalspace3d.org/, 2006.

[20] R. Wieting. Hybrid high-level nets. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 848–855, 1996.

[21] J. S. Willans and M. D. Harrison. Prototyping pre-implementation designs of virtual environment behaviour. In *EHCI '01: Proceedings*

of the 8th IFIP International Conference on Engineering for Human-Computer Interaction, pages 91–108, London, UK, 2001. Springer-Verlag.

[22] C. A. Wingrave and D. A. Bowman. Chasm: Bridging description and implementation of 3d interfaces. In *Proceedings of the Workshop on New Directions in 3D User Interfaces*, pages 85–88, 2005.

[23] X3D. Web3d consortium - royalty free, open standards for real-time 3d communication. Available on: http://www.web3d.org/, 2006.

[24] J. Ying. An approach to petri net based formal modeling of user interactions from x3d content. In *Web3D '06: Proceedings of the eleventh international conference on 3D web technology*, pages 153–157, New York, NY, USA, 2006. ACM Press.

[25] J. Ying and D. Gracanin. Petri net model for subjective views in collaborative virtual environments. In A. Butz, A. Krüger, and P. Olivier, editors, *Smart Graphics*, volume 3031 of *Lecture Notes in Computer Science*, pages 128–134. Springer, 2004.

[26] yWorks the diagramming company. Yed - java graph editor. Available on: http://www.yworks.com/products/yed/, 2006.

B Pacote de Classes Desenvolvido

Conforme abordado na Seção 4.4 deste trabalho, um conjunto de classes C++ foi desenvolvido para a implementação dos modelos que representam as tarefas de interação como RdP. Com estas classes, é possível simular uma RdP, de forma a controlar seu fluxo de dados e facilitar a criação de componentes reutilizáveis por outras redes.

As próximas subseções apresentam um Manual do Usuário, descrevendo as funcionalidades dos métodos e atributos de cada classe. A descrição está baseada no Diagrama de Classes UML apresentado pela Figura 4.5, já ilustrada no decorrer do Capítulo 4.

B.1 Classe Object

A classe *Object* é responsável por determinar nomes e tipos para cada um dos objetos que compõe a RdP. *Object* herda características da classe *qmetaobject* do Qt, uma vez que utiliza-se dos recursos oferecidos pelo mecanismo de *signals* e *slots*. Ela também é a classe base para todas as demais classes do pacote - logo, todos seus métodos são públicos.

Os métodos pertencentes a esta classe são:

- **string getName():** retorna o nome do objeto;
- **void setName(string name):** atribui um nome para o objeto;
- **string getClassName():** retorna o nome da classe do objeto, definido pelo desenvolvedor;
- **void setClassName(string name):** atribui um nome de classe para o objeto.

Já os atributos, todos privados, são:

- **_name:** armazena o nome do objeto;
- **_className:** armazena o nome de classe do objeto.

B.2 Classe InteractObj

A classe *InteractObj* mantém a lista de conectores de entrada e saída para cada objeto do tipo *Place* ou *Transition*, permitindo o controle de criação destes. Para as classes derivadas, ela também oferece a procura de determinados objetos na lista, bem como a captura do conteúdo de cada um dos conectores. Seu destrutor, além de suas atribuições básicas, pode ter como tarefa eliminar um conector de uma lista de conectores (passado como parâmetro).

Os métodos públicos pertencentes a esta classe são:

- **virtual void run():** método abstrato, sem implementação na classe base. Este método define uma interface para as classes que derivam de *InteractObj*;

- **void addConnIn(string name, string className):** permite requisitar à *InteractObj* a criação de um conector de entrada para um objeto da RdP. Como parâmetro, esta função-membro exige um nome para o objeto conector, e um nome de classe, do qual ele fará parte;

- **void addConnOut(string name, string className):** de forma similar ao método anteriormente citado, esta função-membro permite requisitar a *InteractObj* a criação de um conector de saída para um objeto da RdP, passando como parâmetro um nome para o objeto, e um nome de classe;

- **ConnectorIn* findConnIn(string name):** retorna um conector de entrada, após procurar por seu nome na lista de conectores de entrada do objeto referenciado;

- **ConnectorOut* findConnOut(string name):** retorna um conector de saída, após procurar por seu nome na lista de conectores de saída do objeto referenciado;

- **void externalEmit(Token *tk):** permite acessar *signal sendData* para envio de dados emitidos por algum dispositivo ou pela aplicação (que não fazem parte da RdP). É utilizado somente para atribuir valores para as marcações iniciais.

Os métodos protegidos pertencentes a esta classe são:

- **vector<Token*> acquireData():** retorna um vetor de marcações válidas, em um determinado estado da rede;

- **vector<string> acquireNames():** retorna um vetor de caracteres com os nomes de todas as marcas que continham dados válidos, em um determinado estado da rede. Sempre está sincronizada com a função-membro *acquireData*.

Os métodos privados pertencentes a esta classe são:

- **void sendData(Token *tk):** método *signal*, implementado automaticamente pelo Qt. Este *signal* é emitido apenas quando o método *externalEmit* é invocado;

- **bool verifyConnIn(string name):** retorna 1 quando um conector de entrada, de mesmo nome e pertencente a um mesmo objeto, é encontrado na lista de conectores de entrada. Caso contrário, retorna 0;

- **bool verifyConnOut(string name):** semelhante a *verifyConnIn*, retorna 1 quando um conector de saída, de mesmo nome e pertencente a um mesmo objeto, é encontrado na lista de conectores de saída. Caso contrário, retorna 0;

- **void addConnectorIn(ConnectorIn *cIn):** adiciona um conector de entrada a lista de conectores de entrada de um determinado objeto;

- **void addConnectorOut(ConnectorOut *cOut):** adiciona um conector de saída a lista de conectores de saída de um determinado objeto;

- **void removeConnectorIn(ConnectorIn *cIn):** remove um conector de entrada a lista de conectores de entrada de um determinado objeto;

- **void removeConnectorOut(ConnectorOut *cOut):** remove um conector de saída a lista de conectores de saída de um determinado objeto.

Os atributos protegidos pertencentes a esta classe são:

- **list<ConnectorIn*> connectorInList:** lista que armazena todos os conectores de entrada de um objeto da RdP;
- **list<ConnectorOut*> connectorOutList:** lista que armazena todos os conectores de saída de um objeto da RdP.

B.3 Classe Token

A classe *Token* é responsável por encapsular dados primitivos, estruturas de dados, objetos geométricos ou qualquer outro tipo de objeto em marcas, que trafegarão posteriormente na RdP. Dispõe de dois construtores: o primeiro, *default*, é apropriado para objetos *Tokens* temporários, criados e eliminados dentro de um elemento da RdP. Para marcações que irão percorrer a rede, deve-se usar o construtor que recebe como parâmetro o nome da classe do qual determinada marca fará parte (*Token::Token(string className*), definindo o tipo da marcação.

Essa classe possui dois métodos accessors, ambos públicos. São eles:

- **void* getData():** retorna um ponteiro para *void*, indicando o endereço de memória que contém o valor de referência do objeto;
- **void setData(void* dt):** atribui a uma variável o ponteiro do objeto referenciado.

Já o atributo privado é:

- **void* _data:** armazena um ponteiro que contém referência a um objeto de qualquer tipo.

B.4 Classe PetriNet

PetriNet é a classe responsável por simular a rede. Ela possui uma lista que pode armazenar, em ordem seqüencial, os elementos (nodos da RdP) a serem executados. Além disso, dispõe de métodos que definem as marcações iniciais, realizam a conexão entre elementos da rede e atualizam marcações já existentes, a cada nova simulação da RdP.

Para tanto, oferece os seguintes métodos públicos:

- **void run():** tem a função de executar a RdP global, acionando os métodos *run* de seus elementos. Para tanto, estes elementos encontram-se organizados em uma lista de objetos;
- **void addElement(string name, InteractObj *element):** permite inserir um novo elemento a RdP. Recebe como parâmetro o nome do nodo e o objeto que o representa;
- **void removeElement(string name):** permite remover um elemento da RdP, a partir do nome do nodo correspondente;
- **InteractObj* getElement(string name):** retorna um objeto do tipo *InteractObj* da lista da RdP, fazendo uma busca pelo nome do elemento;
- **void changeElement(string name, InteractObj *newElement)):** realiza a troca de um elemento da rede por um novo, sobreescrevendo-o. Recebe como parâmetros o nome do objeto atual e o novo objeto. Durante a substituição, atribui para o novo objeto o nome do objeto substituído;

- **bool insertToken(void* value, Token *tk, InteractObj *objReceiver, string *nameCIn):** através da função-membro *externalConnect*, permite ligar um determinado dispositivo ou aplicação com a RdP, definindo valores para as marcas da RdP. O parâmetro *objReceiver* deve ser, obrigatoriamente, um objeto da classe *Place* ;

- **void updateToken(void* value, Token *tk):** quando chamado, tem por função atualizar as marcações da RdP;

- **static bool connectPN(InteractObj *objSender, string *nameCOut, InteractObj *objReceiver, string *nameCIn):** sua função é conectar um objeto da RdP, definido como origem, a um outro objeto da RdP, definido como destino, através de seus conectores de entrada e saída. Sua utilização simboliza a representação de um arco da RdP. Para tanto, precisa receber como parâmetro um elemento da RdP e o nome de um de seus conectores de saída, além do outro elemento com o nome de um de seus conectores de entrada. Em baixo nivel, faz chamada ao método connect do Qt;

- **static bool disconnectPN(InteractObj *objSender, string *nameCOut, InteractObj *objReceiver, string *nameCIn):** permite desconectar dois elementos, eliminando a representação de um arco da RdP. Opera de acordo com os mesmos parâmetros da função *connectPN*.

Existem também métodos privados, quais são:

- **void verifyPNList(string name):** verifica se já existe um elemento inserido na RdP global com o mesmo nome;

- **void externalConnect(Token *tk, InteractObj *objReceiver, string *nameCIn):** faz a ligação entre um determinado dispositivo ou aplicação com a RdP. Isto permite com que marcas sejam atualizadas a cada novo ciclo de execução da rede, sem que objetos externos que representam dispositivos ou aplicação façam parte efetiva da RdP.

A classe também dispõe de três atributos. O primeiro, de caráter protegido, é:

- **list<InteractObj*> petriNetList:** lista responsável em armazenar todos os elementos de uma RdP. Esta lista pode conter sub redes, além de elementos que representam nodos da rede.

Já os atributos privados pertencentes a classe são:

- **vector<InteractObj*> _externalObjList:** contém a lista de objetos definidos como "externos" pela metodologia. São considerados "externos" aqueles objetos que armazenam informações vindas de dispositivos de entrada, ou fornecidos pela aplicação;

- **vector<Token*> _externalTkList:** contém os dados fornecidos pelos objetos "externos", de maneira encapsulada.

B.5 Classe ConnectorIn

A classe *ConnectorIn* é responsável em reter diferentes tipos de marcas nos conectores de entrada de cada um dos elementos da RdP. Também tem como finalidade verificar se uma determinada marcação pode ficar armazenada em um determinado conector de entrada. Sob o

ponto de vista de *signals* e *slots*, seus objetos geralmente atuam como receptores de marcações da rede.

Os métodos públicos pertencentes a esta classe são:

- **void dispatch():** tem a função de "limpar" um conector de entrada, atribuindo um valor nulo (zero) para sua antiga marca antes dela ser atualizada pelo mecanismo de *signals* e *slots*;
- **void receiveData(Token *tk):** funciona basicamente como um *setToken*, atribuindo um valor para a marca armazenada no conector de entrada. É interpretado pelo compilador do Qt como uma função membro *slot*;
- **bool isFull():** retorna 1 se o conector possui uma marcação não-vazia (válida), e que o nome de classe do conector seja igual ao nome de classe da marcação (mesmo tipo);
- **Token* getToken():** retorna uma marcação presente em determinado conector de entrada;
- **void setParent(InteractObj *p):** define o objeto-pai de um conector de entrada - se é um determinado lugar (objeto do tipo *Place*) ou uma determinada transição (objeto do tipo *Transition*);
- **InteractObj* getParent():** retorna o objeto-pai de um conector de entrada;
- **void* unPackFromToken():** retorna um ponteiro para *void*, indicando o endereço de memória que contém o valor de referência do objeto.

Já os métodos privados são:

- **void sendData(Token *tk):** método *signal*, implementado automaticamente pelo Qt. Este *signal* é emitido apenas quando sub-redes são criadas;
- **bool validateToken():** verifica o nome de classe de um conector com o nome de classe de uma marcação ali armazenada.

E, como atributos privados, têm-se:

- **InteractObj* _parent:** guarda o objeto-pai do conector de entrada;
- **Token* _token:** guarda a marcação depositada em um conector de entrada.

B.6 Classe ConnectorOut

A classe *ConnectorOut* tem características similares a classe *ConnectorIn*. Ela é responsável em reter diferentes tipos de marcas nos conectores de saída de cada um dos elementos da RdP, além de verificar se uma determinada marcação pode ficar armazenada em um determinado conector de saída. No entanto, sob o ponto de vista de *signals* e *slots*, seus objetos geralmente atuam como emissores de marcações para a rede.

Os métodos públicos pertencentes a esta classe são:

- **void dispatch():** tem a função de disparar o *signal sendData*, transmitindo o valor atual da marca depositada em um conector de saída. Após isto, "limpa" o conector, atribuindo um valor nulo (zero) para sua antiga marca;

- **void receiveData(Token *tk):** utilizado somente quando existem sub-redes pertencentes a uma RdP global, tem por função atribuir um valor para a marca armazenada no conector de saída. É interpretado pelo compilador do Qt como uma função membro *slot*;
- **void setToken(Token *tk):** atribui um valor para a marca armazenada no conector de saída. Similar ao método *receiveData*;
- **Token* getToken():** retorna uma marcação presente em determinado conector de saída;
- **void setParent(InteractObj *p):** define o objeto-pai de um conector de saída - se é um determinado lugar (objeto do tipo *Place*) ou uma determinada transição (objeto do tipo *Transition*);
- **InteractObj* getParent():** retorna o objeto-pai de um conector de saída;
- **void packInToken(void* dt):** atribui a uma variável o ponteiro do objeto referenciado.

O método privado pertencente a esta classe é:

- **void sendData(Token *tk):** método *signal*, implementado automaticamente pelo Qt. Responsável pela propagação das marcações pela rede.

Já os atributos privados são:

- **InteractObj* _parent:** guarda o objeto-pai do conector de saída;
- **Token* _token:** guarda a marcação depositada em um conector de saída.

B.7 Classe Place

A classe *Place* permite a criação de objetos da RdP que representam lugares. Tais elementos podem armazenar marcações em seus conectores, definindo o estado atual da rede.

Nessa metodologia, um objeto *Place* tem por função apenas repassar os dados recebidos para os respectivos nodos de transição a ela ligados. Para tanto, uma implementação *default* do método virtual *run* é disponibilizado no pacote, onde:

- **void run():** faz o roteamento de marcações depositadas em um conector de entrada, para um conector de saída específico, de mesmo tipo. Esta ação é executada desde que existam marcações válidas inseridas nos conectores de entrada.

B.8 Classe Transition

A classe *Transition* permite a criação de objetos da RdP que representam transições. Da mesma forma que a classe *Place*, seus elementos podem armazenar marcações em seus conectores, e encaminhar estas para outros nodos da rede. No entanto, conforme a definição de RdP, uma transição também pode alterar o comportamento da rede, definindo novas marcações ou atualizando marcas já existentes.

Transition é definida como uma classe base pelo pacote, servindo de interface para especializações implementadas pelo usuário. Para tanto, ela dispõe de dois novos métodos virtuais protegidos que variam de acordo com o propósito do desenvolvedor, e a ação que se deseja modelar. São eles, de modo geral:

- **virtual void preProcessingTokens():** responsável em coletar as marcações recebidas pelos conectores de entrada de um objeto *Transition*, convertendo as informações encapsuladas em atributos locais utilizados pelas ações do sistema;

- **virtual void distributeTokens():** responsável em depositar novas ou atualizadas marcações em conectores de saída. Um processo de encapsulamento de dados é realizado, antes de efetuar a distribuição das marcas para os próximos nodos da rede.

Além disso, classes derivadas de *Transition* devem implementar o método público *run*, herdado da classe *InteractObj*. Em linhas gerais:

- **void run():** faz o processamento dos dados recebidos pela transição, determinando o comportamento do sistema a medida que ações vão sendo habilitadas ou desempenhadas. Em nossa metodologia, tarefas de seleção, manipulação e navegação, bem como técnicas de interação, são exemplos de ações que podem ser modeladas como transições.

B.9 Classe PlacePage

A classe *PlacePage* permite a criação de objetos de uma RdP que representam sub-redes (superpágina), onde os elementos localizados nos extremos da rede são representados por lugares. *PlacePage* é uma classe derivada de *PetriNet*, e comporta-se como tal. Ela pode ter agregada a sua lista de elementos qualquer objeto do tipo *InteractObj*: lugares, transições e, inclusive, outras sub-redes.

Por padrão, o pacote oferece a implementação dos seguintes métodos públicos:

- **void run():** caso exista uma conexão entre conectores da própria superpágina, realiza o roteamento de marcações depositadas em um conector de entrada, para um conector de saída específico, de mesmo tipo. Esta ação é executada desde que existam marcações válidas inseridas nos conectores de entrada;

- **void exposeIn(string *thisNameCIn, InteractObj *objReceiver, string *nameCIn):** faz o roteamento de marcações depositadas em conectores de entrada da superpágina para conectores de entrada de elementos da sub-rede;

- **void exposeOut(InteractObj *objSender, string *nameCIn, string *thisNameCOut):** faz o roteamento de marcações depositadas em conectores de saída de elementos da sub-rede para conectores de saída da superpágina.

B.10 Classe TransitionPage

A classe *TransitionPage* desempenha papel semelhante a *PlacePage*. A diferença aqui é que os elementos localizados nos extremos da sub-rede devem ser representados por transições. *TransitionPage* também herda as características de *PetriNet*, e pode agregar elementos do tipo *InteractObj*. Apesar de representar transições, objetos dessa classe não têm definida com função à alteração do comportamento do sistema.

Por padrão, o pacote oferece a implementação dos seguintes métodos públicos:

- **void run():** caso exista uma conexão entre conectores da própria superpágina, realiza o roteamento de marcações depositadas em um conector de entrada, para um conector de saída específico, de mesmo tipo. Esta ação é executada desde que existam marcações válidas inseridas nos conectores de entrada;
- **void exposeIn(string *thisNameCIn, InteractObj *objReceiver, string *nameCIn):** faz o roteamento de marcações depositadas em conectores de entrada da superpágina para conectores de entrada de elementos da sub-rede;
- **void exposeOut(InteractObj *objSender, string *nameCIn, string *thisNameCOut):** faz o roteamento de marcações depositadas em conectores de saída de elementos da sub-rede para conectores de saída da superpágina.