Reúso de Modelos em Redes de Petri Coloridas

Adriano José Pinheiro Lemos

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba - Campus II como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Sistemas de Software

Angelo Perkusich

(orientador)

Campina Grande, Paraíba, Brasil

L554R

LEMOS, Adriano José Pinheiro

Reúso de Modelos em Redes de Petri Coloridas.

Dissertação de mestrado, Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande - PB, Março de 2001.

68 p. Il.

Orientador: Angelo Perkusich

Palavras Chave:

1. Redes de Petri de Alto Nível
2. Engenharia de Software
3. Métodos Formais
4. Reúso de Software

CDU - ~~519.711~~ 004.7(043)

REÚSO DE MODELOS EM REDES DE PETRI COLORIDAS

ADRIANO JOSÉ PINHEIRO LEMOS

DISSERTAÇÃO APROVADA EM 28.02.2001

PROF. ANGELO PERKUSICH, D.Sc
Orientador

PROF. JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc
Examinador

PROFª ELIANA SILVA DE ALMEIDA , Drª
Examinadora

CAMPINA GRANDE – PB

Resumo

Neste trabalho investiga-se a aplicação dos principais conceitos de reúso de software na modelagem formal de sistemas. Mais especificamente, trata-se do reúso de modelos de sistemas de software utilizando Redes de Petri Coloridas. O principal objetivo é contribuir para melhorar o processo de modelagem de sistemas de software complexos, distribuídos e concorrentes. Como resultado, introduz-se atividades de reúso no processo de modelagem, mais especificamente armazenamento e recuperação. Tais atividades são suportadas pela introdução de um método e uma técnica integrados de forma a promover o reúso de modelos.

Abstract

In this work the main concepts of software reuse are applied to the formal modelling activities of a software development process. More specifically, we focus the model reuse of software systems described in Coloured Petri Nets. The main goal of this work is to improve the modelling process of complex, distributed, and concurrent software systems. As a result, we come with the introduction of reuse activities on the modeling process, namely the storage and retrieval of artifacts. This activities are suported by the introductio of a technique, and a method, which integrated, provide a way to reuse Petri Net models.

Agradecimentos

Agradeço a todos os amigos, novos e antigos, que, intencionalmente ou não, me ajudaram ou não me atrapalharam durante o desenvolvimento e redação deste trabalho. Espero poder retribuir-lhes o favor não os atrapalhando, quando for possível, em suas fainas mais complexas. Em particular, agradeço a todos colegas que dividiram o espaço do laboratório de redes de Petri e as angústias dos dias inacabáveis.

Um agradecimento especial ao meu orientador Angelo Perkusich pela presteza, sinceridade e seriedade durante os processos, nem sempre agradáveis, de concepção, implementação e documentação deste trabalho de mestrado.

Finalmente, agradeço a minha mãe e meu pai (Salete e Ribamar) pelo exemplo de vida e pela oportunidade de lhes dar alegrias mesmo que fugazes; e a minha namorada Sophia por salvar um ambiente distinto do acadêmico, resguardando assim a minha sanidade mental.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A *Engenharia de Software* como disciplina que trata de métodos, ferramentas e técnicas para tornar a produção de sistemas de software financeiramente eficiente [Cle95], teve seu berço na conferência da OTAN[1] que levou o seu nome em 1968 cujo objetivo era debater possíveis soluções para a chamada *crise de software* - termo também cunhado por ocasião do evento.

Neste evento, entre as soluções propostas, surgiu a da criação de uma indústria de software que tornasse o processo de desenvolvimento de software uma atividade previsível e financeiramente viável, através do uso de processos de desenvolvimento similares aos de outras disciplinas de engenharia bem estabelecidas.

O princípio que norteia a solução supracitada é o reúso de artefatos de software especialmente construídos de forma a permitir que o desenvolvimento de novos sistemas de software não seja feito sempre do "zero", mas sim através da composição de artefatos pré-fabricados. Neste contexto, o conceito de reúso de software foi definido, mais especificamente no artigo do McIllroy [McI69] propondo a criação dessa indústria de software capacitada a produzir, em massa, artefatos de software reusáveis, então chamados de componentes de software.

Passados mais de 30 anos após essa conferência, diversos esforços de pesquisa investem no reúso de software com o objetivo de melhorar os processos de desenvolvimento de sistemas de software [Kru92; MMM95]. De fato, hoje percebe-se que métodos e técnicas de reúso de software encontram-se disseminados entre diversas fases do processo

[1] Organização do Tratado do Atlântico Norte.

de desenvolvimento de sistemas de software com o mesmo objetivo mas atuando em atividades diferentes.

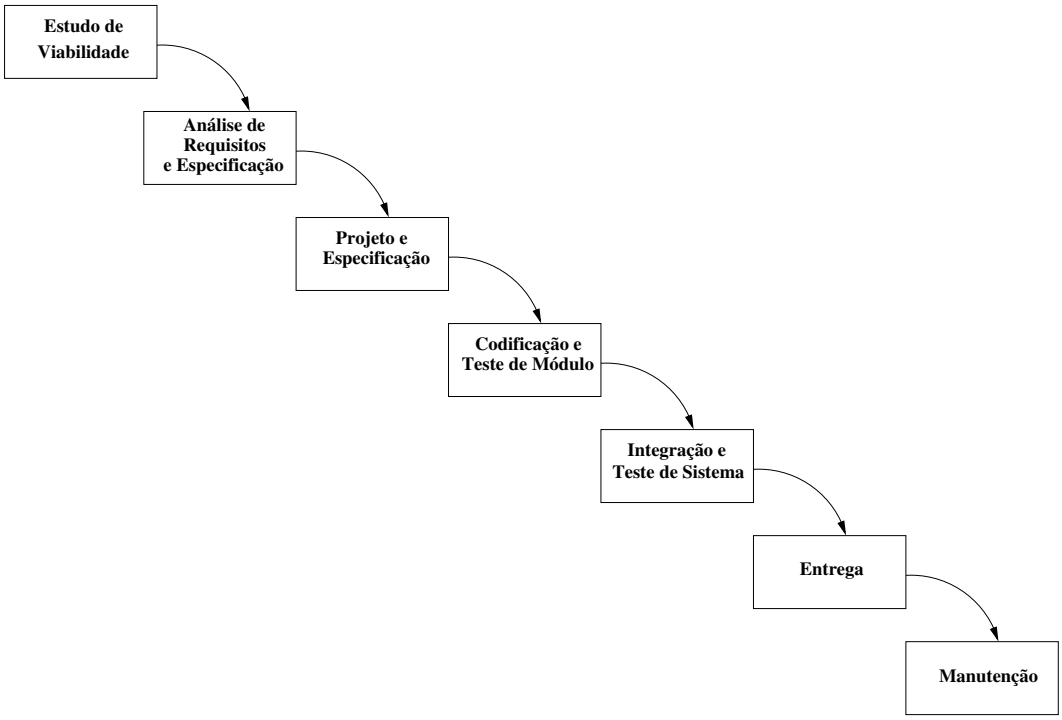


Figura 1.1: Esquema do Modelo Cascata de Produção de Software.

Considere por exemplo o modelo cascata clássico de ciclo de vida do processo de desenvolvimento, ou pelo menos a variante deste discriminada em fases como descrito em [GJM91] e ilustrado na Figura 1.1. Observe que em cada fase da produção de um sistema de software um artefato diferente (especificação, modelos, descrição arquitetural, esquemas de teste, código-fonte) é trabalhado, e que cada um destes artefatos pode ser objeto de reúso dependendo da atividade de desenvolvimento em questão [Fre83]. Não há restrições, tudo que é resultado de esforços de desenvolvimento anteriores pode ser reusado no intuito de melhorar o processo de produção de novos sistemas.

Entretanto, é comum observar-se na literatura e pelas práticas de mercado que o foco principal dos esforços de pesquisa e desenvolvimento em termos de técnicas e métodos de reúso, está na fase de implementação, reusando código fonte ou objeto. Porém, outras fases do processo podem prever atividades de reúso dos artefatos de software afins, enriquecendo com isso o processo de desenvolvimento como um todo.

Um exemplo de que as atividades de reúso se aplicam com sucesso a outras fases do

processo de desenvolvimento de software é o uso, bastante divulgado tanto na comunidade acadêmica quanto na indústria de software, de padrões de projeto [GHJV94]. Na fase de projeto e definição da arquitetura de um sistema cujo processo segue uma metodologia orientada a objetos, essa técnica promove o reaproveitamento do conhecimento especialista a cerca de soluções arquiteturais para problemas recorrentes de projeto. Neste cenário, o objeto de reúso são as descrições de soluções organizadas em um sistema de padrões.

Enfatizando a fase de projeto, onde atividades de modelagem de sistemas de software são aplicadas, métodos e técnicas de reúso podem ser introduzidos com os mesmos propósitos da fase de implementação. Neste caso, o objeto de reúso são os modelos do sistema. Nos casos em que linguagens formais são usadas na modelagem, o objeto de reúso são modelos sobre os quais propriedades desejadas no comportamento do sistema podem ser verificadas através de métodos de análise e simulação [Gar94; GD90].

O uso de métodos formais no processo de desenvolvimento desses sistemas de software ajuda na detecção de inconsistências e falhas no projeto de sistemas antes que estas possam causar grandes perdas financeiras ou mesmo de vidas humanas. Tais ferramentas e métodos embasados em matemática são extremamente úteis quando se busca garantir um maior grau de confiança no funcionamento de um sistema de software quando implementado.

Quando os sistemas de software são complexos[2], as decisões de projeto tornam-se mais difíceis. E quanto mais difíceis são essas decisões, mais extensos e difíceis de manipular os modelos matemáticos se tornam. Técnicas de abstração ajudam a compreender e tratar modelos extensos, mas nem sempre ajudam a reduzir o tempo e o esforço de construção de modelos para novos sistemas. Para tanto, técnicas e métodos de reúso são mais adequados para auxiliar na construção de novos modelos a partir de partes de modelos existentes que sintetizam esforços bem sucedidos em modelagens anteriores.

Com efeito, assim como o reúso de código promove uma melhoria na fase de implementação através da mudança de ênfase do engenheiro, da codificação para integração

[2]Por complexos entende-se neste trabalho grandes, muito extensos.

[Cle95], o reúso de modelos pode promover a diminuição do esforço de modelagem, com conseqüente melhoria de desempenho nas tarefas realizadas na fase de projeto, através do aproveitamento de esforços de modelagem anteriores.

No universo de problemas tratados pela *engenharia de software*, a classe de problemas que é focalizada neste trabalho é a modelagem de sistemas distribuídos e concorrentes. Devido à sua adequação à tarefa de modelagem de tais sistemas [Mur89], o método formal escolhido neste trabalho são as Redes de Petri Coloridas (*Coloured Petri Nets – CPN*) [Jen92].

As *Redes de Petri* são uma ferramenta matemática, especialmente apropriadas à descrição e ao estudo de sistemas de software concorrentes, assíncronos, distribuídos, paralelos, não-determinísticos e/ou estocásticos [Mur89]. As Redes de Petri Coloridas são redes de Alto Nível cuja característica principal é a incorporação da teoria de tipos de dados. Tal característica permite a representação de informações complexas, desta forma, promovendo a descrição mais compacta de modelos. No Capítulo 3 são apresentados os conceitos de Redes de Petri relevantes ao contexto desta dissertação.

O trabalho de modelagem de sistemas de software em Redes de Petri com auxílio de computadores é suportado por alguns ambientes, dentre esses vale citar o PEP (*Programming Environment Based on Petri Nets*) [Gra95; Gra97] e o Design/CPN [CJK97]. Ambos auxiliam na edição gráfica e análise de modelos, entretanto não dispõem de mecanismos que suportem o reúso automático de modelos descritos utilizando tais ambientes. Atualmente, o projetista é forçado a reaproveitar os seus modelos exportando-os de projetos anteriores e importando-os nos novos, prática pouco adequada quando o tempo para conclusão da modelagem é um fator crítico.

Dentre os dois ambientes citados, apenas um suporta o trabalho com a extensão de Redes de Petri que se adequa ao escopo deste trabalho: o Design/CPN. Este ambiente é formado por um conjunto de ferramenta para o desenvolvimento de modelos de Redes de Petri Coloridas. Uma funcionalidade de particular importância para este trabalho é a possibilidade de criar uma hierarquia de modelos, o que é uma facilidade sintática que ajuda na descrição de modelos de sistemas complexos.

1.1 Objetivos da Dissertação

Este trabalho tem como objetivos principais o estudo e aplicação dos princípios de reúso de software na modelagem de sistemas de software complexos, concorrentes e distribuídos utilizando Redes de Petri Coloridas.

Como conseqüência do estudo tem-se uma proposta de solução de reúso de modelos CPN que serve como prova de conceitos. Essa solução é composta por um método de armazenamento de modelos CPN em um repositório e uma técnica para recuperação destes, implementadas na ferramenta Design/CPN. O método e a técnica têm como propósito auxiliar o projetista na construção de novos modelos CPN, promovendo maior eficiência para a tarefa de modelagem formal de sistemas complexos de software.

1.2 Escopo e Relevância

Neste trabalho relata-se um estudo sobre o reúso de software na fase de modelagem de sistemas complexos de software, de acordo com o que foi declarado nos objetivos. Este estudo se norteou pelo objetivo de oferecer mecanismos de reúso de modelos em Redes de Petri Coloridas.

A escolha de Redes de Petri como formalismo se deu devido a sua boa adequação à modelagem de sistemas complexos e concorrentes, bem como a existência de uma boa ferramenta de suporte a modelagem e análise de tais sistemas.

Para a técnica de reúso estabelecida, assume-se que as buscas por modelos resultam em sucesso quando o conjunto de propriedades que os descrevem são atendidas (algum artefato é modelo das especificações declaradas). Esse critério de busca possibilita a seleção de mais de um candidato ao reúso, embora haja sempre a possibilidade de haver uma seleção exata, apenas um candidato seja selecionado.

É importante observar que é necessário anotar os modelos formalmente, por exemplo visando parametrização, desta forma promovendo a integração automática desses em projetos em desenvolvimento.

A importância deste trabalho está na introdução da aplicação de mecanismos sistemáticos de reúso de software na modelagem de sistemas em Redes de Petri Coloridas Hierárquicas [Jen92]. Fundamentalmente, discute-se a aplicação de métodos formais no

contexto de reúso de software, particularmente na fase de modelagem, como atividade principal e não como suporte matemático ao reúso automático de código.

Desta forma, busca-se contribuir para o estabelecimento de métodos e técnicas que promovam um aumento de produtividade no processo de desenvolvimento de sistemas de software através da introdução de mecanismos de reúso na fase de modelagem formal de tais sistemas.

1.3 Estrutura da Dissertação

Esta dissertação está organizada como apresentado a seguir: no Capítulo 2 o tema reúso de software é discutido e o trabalho situado no escopo mais específico de reúso de modelos CPN. No Capítulo 3 são introduzidos os conceitos necessários ao entendimento da solução de reúso de modelos CPN.

No Capítulo 4, a solução de reúso é detalhada com auxílio de um exemplo e alguns aspectos sobre a implementação do método e da técnica que promovem reúso de modelos CPN são levantados de forma mais elaborada. No Capítulo 5 são apresentadas as conclusões e as questões relativas aos trabalhos futuros.

Capítulo 2

Reúso de Software

Neste capítulo, trata-se de reúso no contexto dos processos de desenvolvimento de sistemas de software. Os benefícios da adoção de práticas de reúso são evidenciados, bem como as abordagens de reúso mais conhecidas e algumas de suas classificações. Neste contexto, aborda-se um escopo mais específico de estudo sobre à aplicação de reúso, que se restringe a fase de projeto, especificamente com relação ao uso de métodos formais na construção de modelos de sistemas de software.

2.1 Reúso de Software e Suas Abordagens

A conferência patrocinada pela OTAN[1] no ano de 1968 em Garmisch, [NE68] onde se discutia como tornar a atividade de desenvolvimento de sistemas de software mais eficiente em termos de custos e mais previsível quanto ao tempo de conclusão de projetos, é geralmente aceita como o berço da *engenharia de software* [Kru92]. Nesta conferência, McIlroy [McI69] sugeriu a criação de uma indústria de componentes de software reusáveis, primeira menção ao reúso sistemático de software.

Entretanto, é interessante observar que, historicamente, a preocupação com reúso de software tem raízes anteriores ao nascimento da *engenharia de software*. Já em 1950, como observa Wegner [Weg89], encontram-se preocupações com o desenvolvimento de bibliotecas de sub-programas e, poucos anos mais tarde, observa-se o desenvolvimento de linguagens de alto nível, a exemplo de *Fortran*, que permitiam a definição de

[1] Organização do Tratado do Atlântico Norte.

abstrações de programação tais como funções e procedimentos parametrizáveis.

Reusar esforços bem sucedidos, não importa de que natureza, é economizar tempo e trabalho com conseqüente aumento de produtividade [Nei94; MMM95; HM84]. Essa é uma máxima que tem guiado muitos esforços de pesquisa na busca por melhores práticas de desenvolvimento de sistemas complexos[2] de software nestas últimas três décadas [Hem96; Kru92]. Em seguida apresentam-se, como exemplo, dois casos em que o reúso de software tem proporcionado claros benefícios aos processos de desenvolvimento de sistemas de software complexos.

Como primeiro caso, a generalização e reúso de boas soluções para problemas de projeto que se repetem, reduz o tempo de conclusão e o número de possíveis falhas no sistema causadas por decisões arquiteturais erradas [GAO95]. É disso que trata a aplicação de padrões de projeto [GHJV94], é o estado-da-arte em termos de conhecimento especialista sobre projeto arquitetural de sistemas de software planejados de acordo com metodologias orientadas a objetos. Observe que o ponto central em padrões de projeto é o reúso de conhecimento especialista sobre projetos orientados a objetos.

Um segundo caso é o reúso de código fonte ou objeto. Esta prática pode diminuir significativamente o tempo de codificação de um sistema de software, ou parte dele, e torna menos provável que erros de programação sejam introduzidos na fase de implementação. É pouco provável que um código defeituoso seja propagado através do reúso ao longo do tempo (é mais provável que este código caia em desuso).

Essa prática de reúso é presenciada sob diversas formas, através de componentes de software, onde o reúso é feito de forma estanque, ou seja, código é reusado sem que se conheça os seus detalhes de funcionamento interno, o seu código fonte. Reúso de código sob a forma de *frameworks*, onde código fonte é reusado através de mecanismos de herança em abordagens orientadas a objetos – neste caso, o reúso também implica na aderência à arquitetura de sistema implícita ao *framework*.

Os cenários de reúso descritos acima são apenas dois dos exemplos mais notórios em que a prática de reúso tem um histórico de bons resultados. Pode-se ainda reusar especificações, requisitos coletados, arquiteturas de sistemas, processos, algoritmos e tudo mais que faça parte do processo de desenvolvimento de sistemas de software. De

[2]Por complexos entendemos grande, extenso.

fato, não há restrições quanto ao artefato que se pode reusar, o que determina sua escolha é a necessidade e a fase de desenvolvimento em questão.

Encontra-se uma literatura rica na área de reúso de software no sentido de classificar, de isolar para compreender as diferentes abordagens de reúso. Algumas concentram-se em identificar que artefato é reusado e outras tentam confrontar atividades de reúso em um nível mais abstrato.

Em um dos levantamentos sobre a literatura de pesquisa na área de reúso de software, Krueger [Kru92] particiona as diferentes abordagens de reúso em oito categorias[3] e discute cada uma delas sob quatro pontos de uma taxonomia que ele considera serem comuns a todas as categorias: abstração, seleção, especialização e integração.

Ainda de acordo com Dusink [Dus92], reúso não é feito no "vácuo" mais sim em algum lugar entre as cinco dimensões discriminadas em eixos ortogonais: transformação versus composição, *caixa-preta* versus *caixa-branca*, nível de abstração, produto versus processo e, finalmente, construção de artefatos reusáveis versus aplicação de artefatos reusáveis.

Do ponto de vista metodológico, é fundamental notar que os processos de desenvolvimento de software baseados em modelos de ciclo de vida, implicitamente, consideram que o desenvolvimento de um novo sistema de software sempre começa da estaca zero. Entretanto, é possível adaptar os modelos existentes para que suas fases incorporem atividades específicas ao desenvolvimento de software com reúso. Assim, o processo de desenvolvimento como um todo pode se beneficiar de técnicas, métodos, ferramentas e ambientes operacionais onde o reúso de artefatos é sistematizado e auxiliado por computador (automatizado).

Há alguns trabalhos de pesquisa em que todo um modelo de ciclo de vida de desenvolvimento baseado em reúso foi elaborado [DV94]. Nesses, em cada fase do ciclo de vida estão previstas atividades específicas de reúso. Considere por exemplo, algumas atividades de reúso associadas a fases do modelo cascata de processo de desenvolvimento de software. O resultado é o cenário apresentado na Tabela 2.1.

[3]Linguagens de alto nível, prospecção de código e *design*, componentes em código-fonte, esquemas de software, geradores de aplicações, linguagens de mais alto nível, sistemas de transformações e arquiteturas de software.

Fases do Projeto	**Atividades de Reúso**
Análise	Reúso de requisitos
Projeto e Especificação	Reúso de decisões de projetos (padrões de projetos) e reúso de especificações formais
Codificação e Teste de Módulo	Reúso de código fonte e/ou de código objeto

Tabela 2.1: Exemplo de relação entre fases do modelo cascata e atividades de reúso associadas.

O cenário exemplificado na Tabela 2.1 embora puramente ilustrativo, é útil para identificar algumas das possíveis atividades de reúso, em correspondência direta com as fases nas quais se inserem dentro do contexto de um modelo de ciclo de vida para desenvolvimento de software com reúso. É óbvio que não contempla todas as fases do modelo cascata de ciclo de vida. Para uma visão mais detalhada de um modelo no qual atividades de reúso foram incorporadas em todas as fases veja a documentação do projeto REBOOT [DV94].

Como visto, há um espectro bastante amplo para o estudo da inserção de práticas de reúso de software nos processos de desenvolvimento de software. A próxima seção trata da identificação das atividades relacionadas ao reúso de software e alguns dos percalços que surgem em função dessas atividades.

2.2 Atividades de Reúso de Software

Genericamente, pode-se identificar três atividades que, recorrentemente, estão presentes em qualquer fase de um processo de desenvolvimento que incorpore reúso como prática. São elas: armazenamento de artefatos reusáveis, recuperação de artefatos reusáveis e integração dos artefatos recuperados com ou sem adaptação.

Considerando tais atividades, é comum encontrar-se situações onde se pressupõe a existência de artefatos construídos que possam ser reusados quando necessário for (não há reúso sem que haja um uso prévio). Neste ponto, esbarra-se com mais uma separação de atividades de alto nível frente ao reúso de software: há quem desenvolva

artefatos reusáveis e quem desenvolva a partir de artefatos reusáveis. Embora não haja desenvolvimento baseado em reúso sem desenvolvimento para reúso, neste trabalho focaliza-se a segunda atividade, ou seja, a construção de novos produtos de software a partir de artefatos já construídos e bem usados sem, entretanto, negligenciar as atividades de armazenamento de artefatos.

Neste sentido, duas das atividades recorrentes citadas no início desta seção interessam mais particularmente: como armazenar e como recuperar artefatos reusáveis. A integração pode ser deixada em segundo plano neste primeiro momento.

Recuperar artefatos de software com eficiência não é uma tarefa fácil, depende do quão eficientemente pode-se declarar o que se quer recuperar e como a busca por artefatos que correspondam ao que se declarou é levada a cabo.

Num contexto informal, onde não há como declarar com precisão o que se busca, por exemplo em um repositório FTP de programas, o processo de busca é feito através da identificação dos candidatos através dos seus nomes − que podem ser extremamente inexpressivos tornando a busca frustrante. Já no cenário de busca por uma função em uma biblioteca de funções, dispõe-se das assinaturas das funções, dos seus tipos declarados através dos parâmetros formais. Tais informações são de grande valia, muito embora não seja garantia de que o que se deseja é realmente o que se obteve na busca.

Não há garantias mesmo com as declarações de tipos da assinatura, pelo simples fato de que uma informação puramente sintática sobre um artefato é insuficiente para declarar o que ele faz. Considere por exemplo uma busca por um artefato que implemente uma pilha de números inteiros em uma biblioteca de funções.

Ao declarar-se que o artefato alvo da busca possui duas operações básicas, respectivamente armazenar e recuperar um número inteiro, espera-se ter descrito o comportamento de uma pilha de inteiros. Porém, ter-se-ia tão somente buscado por quaisquer artefatos que implementem um agregado de dados do tipo inteiro. Como resultado, poderia-se obter um artefato que implemente uma fila e não uma pilha, que é um desfecho perfeitamente aceitável em vista do que foi declarado, porém sem utilidade para quem desejava uma pilha a princípio.

Este é apenas um dos problemas associados à recuperação de artefatos para reúso.

Várias pesquisas relacionadas aos problemas de recuperar código para reúso, ou como queiram, componentes de software para reúso, apresentam esforços na tentativa anotar formalmente código, ou seja, enriquecer componentes de software com descrições matemáticas de seus comportamentos de forma a disponibilizar maiores garantias sobre o que recupera [ZW93; ZW95b; FS97].

2.3 Reúso de Modelos

Na fase de projeto de um sistema de software, a construção de um modelo formal do sistema sendo desenvolvido pode ser concluído mais rapidamente caso o projetista disponha de partes de modelos anteriores que possam ser reusados eficientemente no projeto em andamento. Neste contexto, a busca por tais modelos reusáveis pode ser muito mais eficiente que uma simples prospecção de pedaços de modelos dentro de projetos inteiros já concluídos. Isso se deve ao fato do artefato objeto de reúso se tratar de um modelo matemático que pode ter o seu comportamento investigado de forma automática, fato este que facilita bastante o reúso.

Qualquer projetista que já tenha copiado e colado seus próprios modelos de um projeto para o outro é um usuário em potencial de uma técnica ou método ou ferramenta de reúso de modelos. É muito prático guardar esforços de modelagem que poderão facilmente ser reaproveitados num próximo projeto. Essa crença é reforçada pela prática de modelagem e ensino experimentada pelos pesquisadores e professores integrantes do Laboratório de Pesquisa em Redes de Petri (Labpetri) do Departamento de Sistemas e Computação, Campus II da Universidade Federal da Paraíba, e é apoiada pelo desenvolvimento de uma técnica e um método que serão descritos em detalhes no Capítulo 4.

O reúso de modelos formais não é uma idéia inteiramente nova, sendo possível encontrar alguns trabalhos que, direta ou indiretamente, concorrem para adoção de práticas de reúso nas atividades de modelagem formal de sistemas de software.

Trabalhos relacionados a introdução de conceitos e mecanismos de orientação a objetos em Redes de Petri, são passos importantes em direção ao reúso de modelos através do uso de mecanismos de herança e encapsulamento [Lak95; AB96;

LHCB98], alguns deles desenvolvidos dentro do ambiente de pesquisa do Labpetri [Gue97; SCP98]. Em outro trabalho, questões de reúso são abordadas através da proposta de desenvolvimento de um sistema de padrões de modelos de Redes de Petri [NJ99], similar ao conceito de padrões de projeto, a idéia é generalizar soluções de modelagem em Redes de Petri para reúso sistemático posterior.

Contudo, os trabalhos supracitados embora diretamente relacionados, promovem o reúso de modelos através de métodos que se denomina de reúso "caixa-branca"[4] (uso de herança), em contraposição à abordagem de reúso deste trabalho, que é voltada para o reúso "caixa-preta"[5],onde o modelo pode ser reusado da forma como foi recuperado, sem que o projetista necessite fazer grandes alterações. Ainda sim, ambas as abordagens pertencem ao mesmo grupo de idéias e podem ser bem aplicadas concomitantemente.

O tipo de abordagem de reúso de modelos, necessita que se identifique quais as atividades de reúso associadas ao processo de modelagem formal de sistemas de software, e como estas podem ser desempenhadas da forma mais automática possível, com vistas a facilitar o trabalho de modelagem. No que segue, discute-se que atividades de reúso podem ser integradas ao processo de modelagem formal e como aplicá-las.

2.4 Atividades de Reúso na Modelagem Formal de Sistemas de Software

Na fase de projeto do sistema de software, o projetista é o especialista que detém conhecimento sobre o formalismo escolhido para aquela fase do processo de desenvolvimento e, provavelmente, das linguagens de programação utilizadas. Assim, é razoável esperar que o processo de construção dos modelos do sistema sofra influências da metodologia de programação, ou do paradigma adotado (orientação a objetos ou estruturado, por exemplo) bem como do ambiente no qual o projeto será codificado (UNIX, Windows, etc). Estas influências afetam o modo como o modelo deste sistema é construído.

Em um ambiente em que o modelo de um sistema não será desenvolvido do zero, mas sim a partir de modelos reusáveis, o modo como este modelo é construído é também

[4]tradução do termo "white-box".

[5]tradução do termo "black-box".

influenciado pela abordagem de desenvolvimento baseada em reúso. Logo, o projetista muda o foco da atividade principal que é construir o modelo, para a atividade de montar o modelo a partir de pedaços já construídos.

Esta simples mudança de foco, em si, já implica em que o projetista deve então se preocupar em como buscar as partes reusáveis de que necessita para construir um novo modelo. O projetista agora modela por diferença, primeiro ele deve pensar no que vai construir, depois fragmentar a solução, depois ele deve se perguntar que partes da solução podem já ter sido modeladas anteriormente. Em seguida, ele se concentra em como descrever os modelos que deseja recuperar para então, de posse de modelos reusáveis recuperados, concluir seu modelo através da integração das partes novas com as partes recuperadas.

É fácil perceber que, durante a modelagem de cada novo sistema, um projetista pode identificar um bom candidato a modelo reusável e armazena-lo no repositório. Esta atividade deve ser extremamente encorajada e suportada na fase de projeto por ser esta a forma pela qual o repositório de modelos reusáveis se torna mais rico. Tal atividade pode ser automatizada por uma técnica de forma a facilitar e sistematizar a evolução do repositório de modelos.

Ficam assim elencadas as atividades de reúso que passam a fazer parte integral da fase de projeto de sistemas de software, considerando a modelagem formal destes sistemas como processo principal:

1. Identificação de que consta o modelo do sistema (identificação das partes)
2. Seleção de que partes devem ser construídas do zero e que partes podem ser recuperadas para reúso
3. Descrição e busca no repositório
4. Integração dos modelos recuperados ao projeto em desenvolvimento
5. Identificação de modelos candidatos ao armazenamento no repositório de modelos e inclusão dos mesmos no repositório

Muito embora as atividades supracitadas tenham sido descritas de forma bastante abstrata, sua execução prática não foge muito ao que foi descrito. As quatro primeiras

atividades ocorrem na seqüência delineada acima, entretanto, a atividade de atualização do repositório pode ocorrer a qualquer momento durante a modelagem do sistema, não prescindindo de qualquer passo anterior.

A atividade de recuperação de modelos, mais especificamente a descrição dos modelos a serem recuperados, requer uma maior atenção para que este texto não pareça leviano, apresentando uma enumeração das atividades como se fossem simples e óbvias.

Descrever o que se deseja de um modelo formal pode apresentar os mesmos problemas descritos na Seção 2.2 com a situação exemplo da busca por um artefato que implemente o controle de uma pilha de inteiros numa biblioteca de funções. É necessário determinar-se com clareza o nível de precisão com que a busca por um modelo vai ocorrer.

Pode-se ter uma busca em qualquer dos níveis extremos a seguir. Pode ser suficiente apenas uma busca feita com base em informações sintáticas sobre o modelo (uma sentença de busca que só avalie critério sintáticos) ou, no outro extremo, pode ser interessante recuperar apenas os modelos que atendam a restrições comportamentais (semânticas) equivalentes à própria descrição total do modelo desejado, ou seja, o próprio modelo descrito na sentença de busca.

Ambos os extremos são indesejados, no primeiro caso (critérios puramente sintáticos) não há como garantir que o que se recuperou é o que se deseja reusar. No segundo caso (busca por equivalência comportamental, ou semântica), pode-se ter a situação em que a linguagem usada para escrever a sentença de busca seja a mesma linguagem usada para modelar o sistema, isso pode resultar na construção total de um modelo para a recuperação de outro idêntico do repositório.

Tal situação como descrita pelo segundo caso é pouco provável em virtude das linguagens de especificação orientadas a modelos (Redes de Petri e grafos de estado, por exemplo) não serem adequadas a descrição de propriedades, o que notadamente é o escopo das linguagens de especificação orientadas a declarações (Lógicas, Álgebras e Expressões Quantificadas, por exemplo).

Assim, as buscas devem ser realizadas sob critérios que estejam a meio caminho entre os extremos acima descritos. Vale salientar que quanto mais restrito é o critério que determinará se um modelo é um candidato a reúso, mais extensa é sentença de

busca, mais o projetista descreverá o modelo desejado. A escolha deste nível de precisão na recuperação de modelos pode impactar no desempenho da recuperação, onerando a aplicação de práticas de reúso no processo de modelagem.

Por outro lado, o relaxamento do critério de busca tem reflexos diretos nas atividades de integração dos modelos recuperados. Caso o modelo resultante seja resultado de uma busca mais relaxada, com uma descrição parcial[6], é bastante provável que o modelo recuperado necessite de um grande número de alterações manuais para ser finalmente integrado com sucesso ao modelo em desenvolvimento.

Um aspecto interessante que não pode ser identificado tão claramente, quanto as atividades de reúso elencadas anteriormente, é o fator cultural. A aplicação de práticas de reúso em fases iniciais do ciclo de desenvolvimento de sistemas de software proporciona o benefício colateral de institucionalizar o reúso desde muito cedo na fase concepção do sistema [Tra95]. É a introdução de uma cultura de reúso que torna a produção de sistemas de software uma atividade com caráter de engenharia no sentido próprio da disciplina.

A seguir, tem-se uma descrição dos pontos a serem considerados no desenvolvimento de uma solução de reúso de modelos formais. A solução descrita segue os princípios já introduzidos e é uma guia para a implementação descrita em detalhes no Capítulo 4.

2.5 Uma Solução Para Reúso Sistemático de Modelos

Nesta seção discute-se como os princípios de reúso de software podem ser aplicados a fase de projeto onde uma visão da arquitetura do sistema é traçada. De acordo com o que foi exposto nas seções anteriores, uma boa prova de que o reúso de modelos pode ser aplicado com sucesso seria a existência de procedimentos sistemáticos para armazenamento e recuperação de modelos.

Assim, a seguir, apresenta-se uma solução de reúso suportada por um método de armazenamento de modelos formais descritos em Redes de Petri Coloridas, bem como de uma técnica para recuperação de modelos de forma automática baseada na técnica

[6]Por parcial queremos dizer que a sentença de busca consta de apenas parte nas restrições desejáveis para a recuperação de um candidato ao reúso.

de Verificação Automática de Modelos. Ambos, a técnica de Verificação Automática de Modelos bem como o formalismo de Redes de Petri são descritos Capítulo 3.

2.5.1 Método para Armazenar Modelos Reusáveis

De modo a ilustrar o método, considere a situação de uma criança brincando com um dos brinquedos mais populares do mundo: os blocos Lego. Esta criança reconhece uma coleção de características que um objeto do mundo real tem, por exemplo uma casa, e tenta imitar em escala reduzida aquele objeto usando os blocos Lego. Se prestarmos bem atenção, o mesmo bloco que já foi parte de uma maquete anterior é usado novamente em um contexto diferente porém similar.

A situação supracitada guarda forte analogia com as práticas de projeto de engenharias bem estabelecidas como a civil e elétrica, onde os engenheiros trabalham com esquemas matemáticos que representam as construções fundamentais. Apenas, por serem documentos matemáticos, estes esquemas podem ser estudados, analisados e/ou simulados de forma a oferecer garantias sob o comportamento dos objetos reais antes mesmo de sua construção.

No contexto de uso de Redes de Petri como uma linguagem adequada para a construção de modelos matemáticos de sistemas de software, os blocos "Lego" são, na maioria das vezes, as construções básicas da linguagem, que serão apresentados no Capítulo 3.

Todavia, isso não é uma tautologia. Pode-se formar blocos fundamentais tão mais complexos quanto se desejar e, com esses, construir modelos bastante grandes de forma mais eficiente. Para tanto, após identificados esses blocos, deve-se ter uma forma sistemática de guarda-los para uso futuro (reúso), uma forma sistemática de recuperar o bloco adequado a uma necessidade específica e uma forma de integra-los ao novo modelo que se deseja construir. Não é necessário, contudo, que estes blocos sejam de uma granularidade prefixada, ou mesmo que sejam padrão. Cada projetista pode reconhecer os seus blocos reusáveis dentre os modelos que constrói em domínios específicos.

O estabelecimento de regras para o armazenamento desses blocos, que doravante serão chamados de modelos reusáveis, deve prever as atividades posteriores de recuperação e integração. Assim, alguns dos princípios podem ser enumerados da seguinte forma:

- Os modelos reusáveis devem ser organizados em uma hierarquia que reflita uma separação de domínios de conhecimento (estruturas de dados, algoritmos de coerência de cache, protocolos de comunicação, etc). Desta forma, o espaço de busca pode ser reduzido.

- Uma padronização, baseada no domínio, dos nomes usados nos modelos reusáveis armazenados. Isso permite a parametrização da busca, tornando-a passível de automatização.

- As informações dos tipos de dados contidas no modelo devem ser armazenada de forma a serem facilmente acessadas posteriormente. Isso facilita os processos de integração dos modelos que eventualmente forem recuperados.

- O modelo deve ser armazenado com informações adicionais sobre como este pode ser utilizado, um caso de uso. Isso tem duas finalidades, a primeira é auxiliar na recuperação através da informação do que um ambiente deve esperar do comportamento externo de um modelo. A segunda finalidade é fornecer um exemplo de uso de forma a auxiliar o projetista no momento de integrar o modelo recuperado no projeto em construção.

Não se tem a pretensão que os princípios apontados acima sejam de aplicação geral para o desenvolvimento de métodos ou técnicas de armazenamento de modelos reusáveis. Essas são as guias que foram utilizadas para construir um método de armazenamento de modelos em Redes de Petri Coloridas Hierárquicas.

Seguramente, essas guias são influenciadas enormemente pelas características da linguagem (tipos de dados complexos e o conceito de hierarquia), bem como pelas restrições funcionais da ferramenta de Redes de Petri adotada: o Design/CPN, ferramenta que será descrita no Capítulo 3.

Os detalhes de implementação que envolvem o processo de armazenamento de modelos CPN são detalhados no Capítulo 4, onde um cenário de reúso de modelos é apresentado e nele o método de armazenamento é ilustrado. A seguir, descreve-se de forma sucinta como se pode ter uma técnica para recuperar modelos reusáveis armazenados de acordo com o método descrito nesta seção.

2.5.2 Técnica para Recuperar Modelos Reusáveis

A atividade de busca por um modelo, de acordo com certos critérios entre uma coleção de modelos armazenados, apresenta algumas dificuldades, por exemplo, como descrever as propriedades desejadas do modelo, ou como proceder a busca por candidatos a recuperação. No caso de Redes de Petri, a explosão do tamanho do espaço de estados que representa o comportamento de um modelo é geralmente o maior dos problemas.

Desta feita, enumera-se aqui as guias de uma técnica para recuperar modelos CPN Hierárquicos observando a necessidade de eliminar ou reduzir os problemas intrínsecos a essa atividade, bem como se beneficiando da forma criteriosa com a qual os modelos foram armazenados. Assim é importante:

- Usar uma linguagem adequada a descrição de propriedades dos modelos (no caso modelos CPN). Esta linguagem deve ser passível de integração com Redes de Petri Coloridas de forma a permitir automatização do processo de verificação de propriedades.

- Permitir que um projetista informe, interativamente de preferência, a que domínio sua busca irá se restringir. É neste momento que a ferramenta de edição e análise de CPNs hierárquicas começa a influenciar na usabilidade da técnica, quanto maiores forem as facilidade de interação com o usuário tanto mais prático e fácil será o uso da técnica.

- Verificar as propriedades descritas contra os modelos armazenados de forma automática. Neste momento, os modelos devem poder ser parametrizados de forma que a mesma fórmula possa ser verificada para os diversos modelos definidos no mesmo domínio. É nesta atividade onde a padronização dos nomes exigida pelo método de armazenamento é de maior importância.

- Possibilitar que os modelos sejam exportados juntamente com os seus tipos de dados associados em arquivos específicos de forma que estes possam ser usados em uma etapa posterior de integração.

Na prova de conceito que é o cenário de reúso relatado no Capítulo 4, os detalhes de implementação desta técnicas são apresentados. Atrelado a isso, o objetivo do próximo

capítulo é descrever de forma bastante sucinta os métodos, as linguagens, as técnicas e ferramentas necessárias a compreensão da solução de reúso de modelos proposta neste trabalho.

Capítulo 3

Conceitos Básicos

Neste capítulo estão descritos os conceitos básicos mais importantes ao entendimento do restante deste trabalho. Primeiramente, serão introduzidas as redes de Petri, formalismo base desta pesquisa. Em seguida, descreve-se a técnica de Verificação Automática de Modelos, usada na implementação da técnica de recuperação de modelos CPN detalhada no Capítulo 4. Finalmente, descreve-se a Lógica Temporal Ramificada (CTL) e sua variante, o ASK-CTL, usada para descrever propriedades sobre o comportamento dos modelos CPN.

3.1 Redes de Petri

Os conceitos que serão expostos nesta seção servem apenas a construção de uma base para o entendimento deste trabalho. Aqui, serão apresentadas as redes de Petri Lugar/Transição e Coloridas, respectivamente nos contextos de redes de *Baixo* e *Alto Nível*.

De acordo com Murata [Mur89], redes de Petri são uma ferramenta matemática, especialmente apropriadas à modelagem e ao estudo de sistemas de software concorrentes, assíncronos, distribuídos, paralelos, não-determinísticos e/ou estocásticos. De fato, as redes de Petri podem ser aplicadas à modelagem de qualquer sistema com tais características.

Além de sua notação matemática, as redes de Petri possuem uma representação gráfica associada que torna mais fácil a interação entre os indivíduos envolvidos na

construção de um sistema de software.

Uma rede de Petri é composta de uma *estrutura de rede*, *inscrições* associadas a essa estrutura e uma *marcação*. A estrutura da rede e as inscrições definem a sintaxe de uma rede de Petri. A evolução de suas marcações, segundo uma *regra de ocorrência*, estabelece a sua semântica.

Uma estrutura de rede de Petri é uma tripla $N = \langle P, T, F \rangle$, na qual:

- P é um conjunto finito de lugares;
- T é um conjunto finito de transições;
- $F \subseteq P \times T \cup T \times P$ é uma relação de fluxo;
- $P \cap T = \varnothing$.

Note que, de acordo com a relação de fluxo F, os arcos da estrutura sempre conectam nós de tipos diferentes. Graficamente, os lugares da estrutura de uma rede de Petri correspondem a círculos, as transições a retângulos e a relação de fluxo a arcos direcionados.

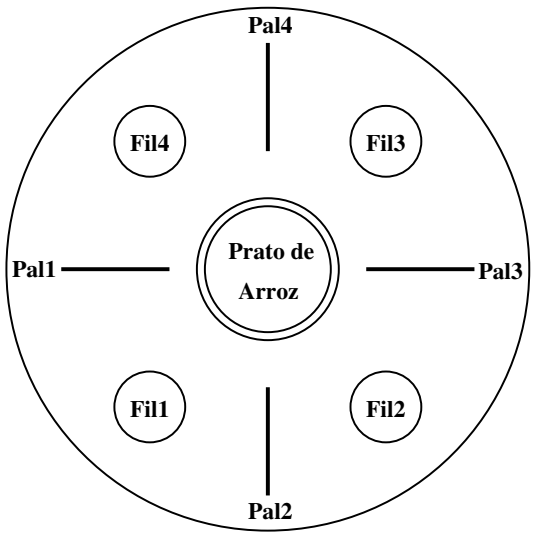


Figura 3.1: Ilustração do sistema do jantar dos filósofos.

Na Figura 3.1, ilustra-se o clássico sistema imaginado por Dijkstra onde filósofos chineses jantam dividindo um prato de arroz em torno de uma mesa, ao longo da qual estão dispostos palitos em número igual a quantidade de filósofos. Neste exemplo,

cada filósofo pode estar em um de dois estados possíveis: comendo ou pensando. Inicialmente, todos os filósofos estão pensando e, para que qualquer um deles comece a comer, é necessário que se pegue dois palitos ao mesmo tempo – o da esquerda e o da direita. Na Figura 3.2 ilustra-se o modelo em redes de Petri do jantar posto para quatro filósofos.

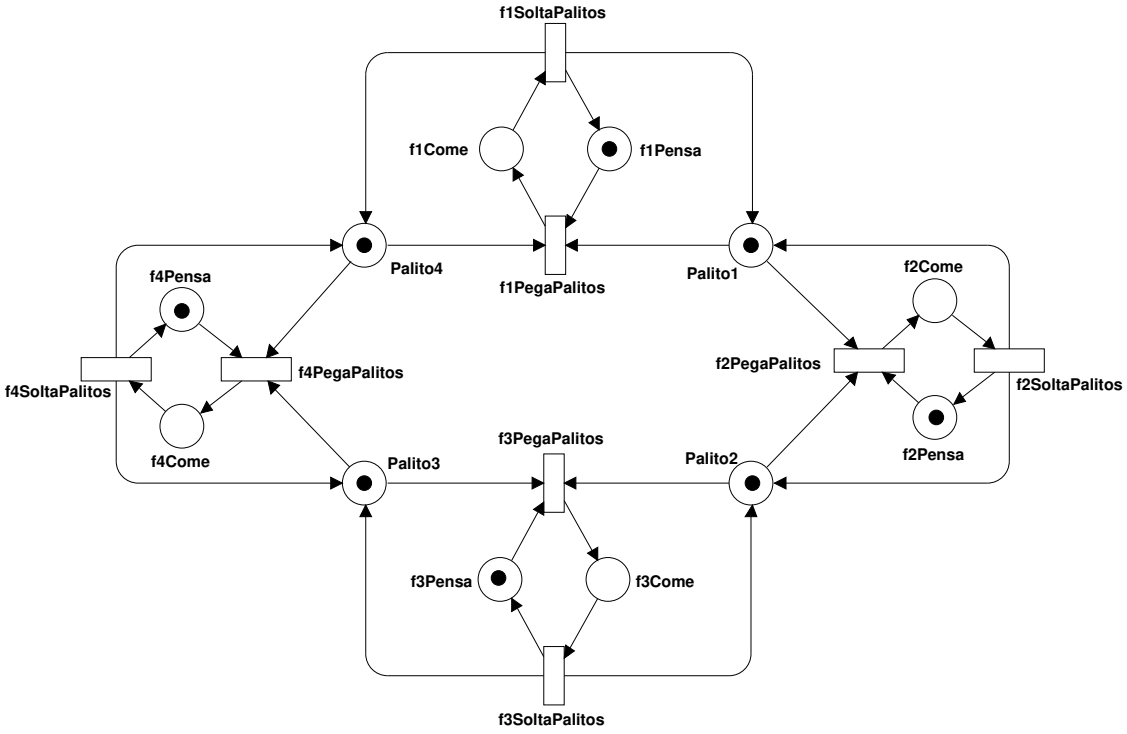


Figura 3.2: Rede de Petri Lugar/Transição, modelo do jantar dos filósofos.

Em uma rede de Petri, os pontos pretos, chamados *fichas*, dentro dos lugares indicam o estado em que se encontra o sistema. Assim, no modelo do jantar dos filósofos na Figura 3.2, as fichas depositadas nos lugares **f(1)(2)(3)(4)Pensa** e **Palito(1)(2)(3)(4)** indicam que todos os filósofos estão pensando e que todos os palitos estão em cima da mesa. Esta associação de fichas aos lugares de uma rede chama-se marcação, e estabelece o estado de uma rede de Petri. Embora o exemplo de rede de Petri da Figura 3.2 não deixe claro, há dois tipos de inscrições para esta rede de Petri: o *peso dos arcos* (que no caso foi suprimido por ter valor unitário) e a *inicialização*, que é a marcação inicial da rede.

Para o que foi discutido até o momento, utilizou-se um modelo de rede de Petri Lugar/Transição, que pertence à classe de redes de Petri de Baixo Nível. Nas re-

des de Petri de Petri de Baixo Nível (como a Lugar/Transição) as fichas representam informações meramente binárias, indicadas pela sua presença ou ausência nos lugares.

O comportamento de uma rede de Petri é determinado pelo conjunto de estados alcançáveis a partir de uma marcação inicial, pela ocorrência ou disparo de transições. Um conjunto de regras denominado *regra de ocorrência ou disparo* determina precisamente em que condições uma transição está habilitada a ocorrer e, quando ocorrer, o que precisamente acontece na rede.

Esta regra de ocorrência pode definir como a rede ilustrada na Figura3.2 muda de estado. Antes porém, de definir o funcionamento da regra de ocorrência, é necessário que alguns conceitos básicos sejam estabelecidos:

Lugares de entrada: lugares de onde partem os arcos direcionados (*arcos de entrada*) que chegam a uma transição;

Lugares de saída: lugares para onde partem os arcos direcionados (*arcos de saída*) que saem de uma transição.

Veja então, a descrição da regra de ocorrência para as redes de Petri Lugar/Transição, tipo descrito até o momento:

- Uma transição está habilitada se os seus lugares de entrada contiverem fichas na quantidade necessária descrita pelos respectivos pesos de seu arcos de entrada;
- Uma transição pode ocorrer caso esteja habilitada;
- A ocorrência ou disparo de uma transição resulta na retirada do número de fichas, especificado nos arcos de entrada, dos respectivos lugares de entrada e, no depósito de fichas na quantidade especificada pelos arcos de saída, nos respectivos lugares de saída.

Desta forma, observando o exemplo do sistema dos filósofos, no modelo apresentado pela Figura 3.2, as transições **f(1)(2)(3)(4)PegaPalitos** estão todas habilitadas no momento inicial. A ocorrência, por exemplo, da transição **f1PegaPalitos** ocasiona a retirada das fichas dos lugares **Palito1** e **Palito4**, e o depósito de uma ficha no lugar **f1Come**.

O conjunto das marcações alcançáveis ou estados possíveis a partir de uma marcação inicial pode ser representado por um grafo denominado *grafo de ocorrência* onde cada

nó representa uma marcação alcançável e cada arco indica a transição que deve ocorrer para alcançar um novo estado a partir de um antecessor. Logo, o grafo de ocorrência de uma rede de Petri é uma representação do seu espaço de estados, uma codificação do seu comportamento.

Assim, pode-se estudar o comportamento de um sistema modelado com redes de Petri através da exploração do seu espaço de estados, investigando o seu grafo de ocorrência. Observe que este grafo pode ser gerado completa ou parcialmente, dependendo do modelo sendo investigado. Isso se deve ao fato de que uma rede de Petri pode ter infinitos estados não interessando assim a geração completa de seu grafo de ocorrência para propósitos práticos.

3.1.1 Redes de Petri Coloridas

As redes de Petri Lugar/Transição são úteis para a modelagem e estudo de sistemas bastante simples. Quando a complexidade dos sistemas a serem modelados aumenta, surgem algumas restrições ao seu uso. Entre as restrições, pode-se citar a necessidade de duplicação da estrutura de rede para modelar processos semelhantes ou idênticos. Isso ocorre devido ao fato de ser impossível diferenciar os processos por meio de fichas.

Dessa forma, visando suprir tais limitações e facilitar a modelagem de sistemas complexos, extensões foram propostas para as redes da classe de Baixo Nível. Dentre elas, tem-se a classe das *redes de Petri Temporais* e a classe das redes de Petri de Alto Nível. As redes de Petri de Alto Nível caracterizam-se sobretudo pela incorporação da teoria de tipos de dados. As fichas para estas redes podem carregar informação complexa, que é manipulada pelo uso de uma linguagem.

O fato das fichas expressarem informações complexas aumenta o poder de descrição dessas redes e, conseqüentemente, modelos mais compactos podem ser obtidos. Essa flexibilidade na manipulação da informação permite ao projetista distribuir a complexidade do modelo de um sistema entre as inscrições e a estrutura da rede. Dentre as redes de Petri de Alto Nível, destacam-se as redes de Petri Coloridas [Jen92].

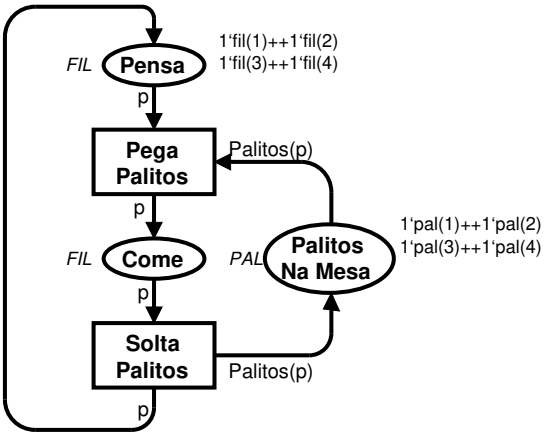
Uma rede de Petri Colorida compõe-se de três partes distintas: *estrutura*, *declarações* e *inscrições*. A estrutura é formada por lugares, transições e arcos direcionados, tal como descrito para as redes de Petri Lugar/Transição. As declarações definem con-

juntos de cores (domínios), variáveis e operações (funções) usadas nas inscrições. As inscrições, por sua vez, podem ser de quatro tipos:

1. *Cores dos Lugares:* determinam a cor associada ao lugar. Um lugar só pode comportar fichas cujos valores respeitem sua cor;
2. *Guardas:* são expressões booleanas que restringem a ocorrência das transições;
3. *Expressões dos Arcos:* servem para manipular a informação contida nas fichas;
4. *Inicializações:* associadas aos lugares, estabelecem a marcação inicial da rede.

Veja na Figura 3.3, o modelo em redes de Petri Coloridas do sistema dos filósofos descrito pela rede Lugar/Transição mostrada na Figura 3.2. Observe que as partes do modelo com estruturas redundantes foram "dobradas" compactando o modelo. Os filósofos e os palitos foram representados por tipos de dados diferentes através do uso de fichas com as cores **FIL** e **PAL**. Neste modelo, note que a inscrição que indica a marcação inicial do lugar **Pensa** é da forma **1‘fil(1) ++ 1‘fil(2) ++ 1‘fil(3) ++ 1‘fil(4)**, onde o **1** antes de cada **fil(...)** indica que há apenas um filósofo deste tipo naquele lugar. Isso porque o mesmo lugar pode conter várias fichas de um mesmo valor, assim essas expressões usam *multiconjuntos* para inscrever os lugares. Um multiconjunto é um conjunto no qual pode haver repetição de elementos. O leitor interessado nas definições formais de redes de Petri Coloridas pode encontra-las em [Jen92].

Como pode ser constatado na Figura 3.3, as inscrições e as declarações de cores usadas na rede são feitas usando-se uma linguagem, neste caso uma linguagem funcional, SML'97 [Ull93]. Nas declarações abaixo do modelo CPN dos filósofos, encontram-se as definições das cores (tipos) que representam os filósofos (**FIL**) e os palitos (**PAL**). Ambas as cores podem ter elementos nomeados **fil** e **pal** indexados por inteiros entre 1 e n, que no caso é 4. Assim, o lugar **Pensa** pode conter, em um determinado momento, o segundo filósofo indicado pela inscrição **1‘fil(2)** e o lugar **Palitos Na Mesa**, o segundo palito indicado pela inscrição **1‘pal(2)**.A função **Palitos** criada realiza a retirada dos palitos situados a esquerda e a direita de um determinado filósofo passado como parâmetro. A variável **p** é criada para assumir o valor de filósofo durante a ocorrência de uma transição.




```
val n =4;
color FIL = index fil with 1..n declare ms ;
color PAL = index pal with 1..n declare ms ;

var p : FIL ;

fun Palitos(fil(i)) = 1‘pal(i)++1‘pal(i mod n + 1) ;
```

Figura 3.3: Modelo em redes de Petri Coloridas do jantar dos filósofos.

Assim, caso a transição **Pega Palitos** ocorra com **p** assumindo o valor **fil(1)** indicando que uma ficha com este valor esta sendo retida do lugar **Pensa**, a função **Palitos** irá remover duas fichas da cor **PAL** do lugar **Palitos Na Mesa** com os valores **pal(1)** e **pal(4)**. Em seguida, uma ficha de valor **fil(1)** é depositada no lugar **Come** concluindo uma mudança de estado da rede.

É importante observar que a regra de ocorrência (ou disparo) para as redes de Petri Coloridas depende de mais alguns conceitos: variáveis de transição, ligação e elemento de ligação. As variáveis de transição são aquelas encontradas nos arcos direcionados, por exemplo **p**, que é variável da transição **Pega Palitos**. Uma ligação é a associação de uma variável de transição a um valor da sua cor (ou tipo), como exemplo de ligação, pode-se ter $l_1 =< p = fil(1) >$. Um elemento de ligação é um par (transição,ligação). Por exemplo, considerando os exemplos supracitados, $el_1 = (PegaPalitos, l_1 =< p = fil(1) >)$ é um elemento de ligação. Mais uma vez, o detalhamento e formalização destes conceitos é deixado como referência ao leitor interessado [Jen92].

3.1.2 O Design/CPN

O Design/CPN é um pacote de ferramentas para o desenvolvimento de modelos de redes de Petri Coloridas, e é constituído de quatro ferramentas computacionais integradas em um único pacote:

1. Editor gráfico para construir, modificar e executar análise sintática de modelos CPN;
2. Um simulador podendo operar simulação interativa ou automática, possibilitando ainda definir diferentes critérios para parada e observação da evolução da rede;
3. Uma ferramenta para gerar e analisar grafos de ocorrência de modelos CPN;
4. Uma ferramenta para análise de desempenho que possibilita a simulação e observação de dados de desempenho de modelos CPN.

O Design/CPN é um dos pacotes de ferramentas mais elaborados para construção, modificação, e análise de redes de Petri, e tem sido extensivamente utilizado com sucesso em diferentes domínios de aplicação. O leitor interessado em maiores detalhes pode consultar [CJK97].

3.1.3 Introdução Informal a Redes de Petri Hierárquicas

Nesta seção introduzimos informalmente as redes de Petri Coloridas Hierárquicas (*Hierarchical Coloured Petri Nets* – HCPN) [Jen92]. O objetivo é familiarizar o leitor com a nomenclatura utilizada em modelos HCPN. A motivação para definir as HCPNs é disponibilizar mecanismos para construir modelos através da combinação de um conjunto de CPNs denominadas *páginas*. Esta abordagem pode ser comparada à construção de um programa a partir de um conjunto de módulos e funções.

Do ponto de vista teórico uma HCPN possibilita descrever os mesmos tipos de sistemas que uma CPN não hierárquica e, portanto, são modelos equivalentes. É sempre possível transformar uma HCPN em uma CPN não hierárquica. Contudo, as HCPNs disponibilizam ao projetista, mecanismos de abstração que permitem descrever de forma mais organizada modelos de sistemas complexos. Considerando que no caso de

sistemas complexos, um dos grandes problemas enfrentados por desenvolvedores de sistemas atualmente é lidar com muitos detalhes ao mesmo tempo, as HCPNs disponibilizam mecanismos de abstração que permitem ao projetista lidar e manter o foco em uma determinada parte de um modelo de cada vez.

Modelos HCPN são implementados utilizando-se os conceitos de *lugares de fusão* e *transições de substituição*. Lugares de fusão são estruturas que permitem especificar um conjunto de lugares como funcionalmente um único lugar, isto é, se uma ficha é removida ou adicionada em um dos lugares, uma ficha idêntica é adicionada ou removida dos outros lugares pertencentes ao conjunto. Um conjunto de lugares de fusão é denominado de conjunto de fusão (*fusion set*).

Uma transição de substituição pode ser vista como uma transição de mais alto nível que se relaciona a uma CPN mais complexa que descreve com mais detalhes as atividades modeladas pela transição de substituição. A página que contém a transição de substituição é denominada de *superpágina* da CPN mais detalhada correspondente, que por sua vez é denominada de *subpágina*. Cada transição de substituição é denominada de *supernó* da subpágina correspondente.

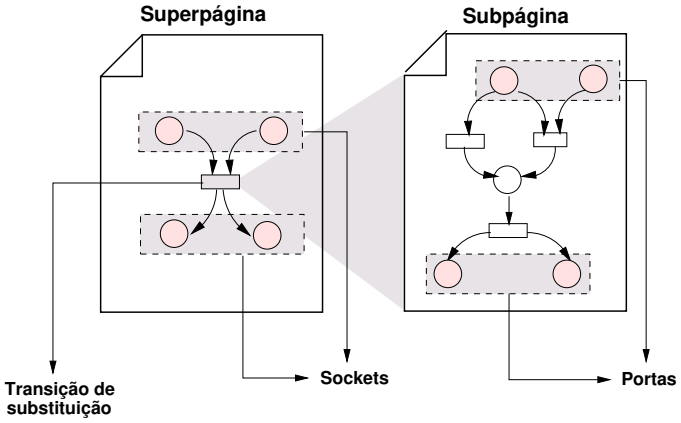


Figura 3.4: Exemplo de uma hcpn com duas páginas.

Uma transição de substituição se relaciona com sua subpágina através da utilização de um tipo de conjunto de fusão de dois membros denominados portas e *sockets*. Estas estruturas descrevem a interface entre a transição de substituição e a subpágina. *Sockets* são atribuídos aos lugares conectados à transição de substituição e portas são associadas a determinados lugares na subpágina tal que um par *socket*/porta formam

um conjunto de fusão. Dessa forma, quando uma ficha é depositada num *socket*, ela aparece também na porta associada àquele *socket*, permitindo assim a conexão entre a superpágina e a subpágina. Cada *socket* pode ser associado a uma ou mais portas, e uma porta pode ser associada somente a um *socket*. Na Figura 3.4 ilustra-se uma hierarquia de duas páginas conforme descrito até o momento.

Como dito anteriormente, é sempre possível traduzir uma rede de Petri hierárquica para sua correspondente não hierárquica. Para isso, basta substituir cada transição de substituição e arcos conectados, por sua respectiva subpágina, "colando" cada *socket* à sua respectiva porta. Detalhes teóricos relacionados a HCPNs podem ser encontrados em [Jen92].

3.2 Verificação Automática de Modelos

Verificação de Modelos é uma técnica automática originalmente desenvolvida para verificar sistemas reativos de estados finitos, tais como projetos de circuitos seqüenciais e protocolos de comunicação [CGL96]. As especificações comportamentais a serem verificadas são expressas em uma lógica temporal proposicional e o sistema é modelado por um grafo de transição de estados. Nesta técnica, um eficiente procedimento de busca é usado para determinar automaticamente se as especificações são satisfeitas pelo grafo de transição de estados, ou seja, se este é um modelo para elas.

A escolha desta técnica em detrimento, por exemplo, da prova mecânica de teorema [CL73] para compor a solução de recuperação de modelos neste trabalho, deveu-se principalmente pelo fato do procedimento de Verificação de Modelos ser automático. O usuário desta técnica só necessita de uma representação de alto nível do modelo do sistema e a especificação a ser verificada. O que acontece é que o verificador de modelos terminará sua execução com um resultado *verdade*, indicando que o modelo satisfaz a especificação, ou um resultado *falso*, seguido de um contra-exemplo, mostrando uma seqüência de execução que explica o por quê da fórmula não ser satisfeita pelo modelo. Este contra-exemplo é bastante útil para encontrar erros de modelagem.

Apesar de suas grandes vantagens, inicialmente, os verificadores de modelos só eram capazes de verificar propriedades em sistemas pequenos devido ao problema conhecido

como *explosão do espaço de estados*. Sistemas cujos modelos apresentavam uma quantidade de estados alcançáveis muito grande, e que crescia de forma não polinomial, tornavam o uso desta técnica impraticável. Isso mudou por volta de 1980 com a descoberta de uma forma de representação das relações de transição chamada de Diagramas de Decisão Binários Ordenados (DDBO)[Bry86], que permitiu a verificação de sistemas realmente complexos.

O algoritmo original de verificação de modelos juntamente com a nova maneira de representação das relações de transições, ficou conhecida como Verificação de Modelos Simbólica [JRBM92]. Esta combinação permite a verificação de sistemas reativos muito extensos, sistemas com mais de 10^{120} estados por exemplo [CGL94].

O funcionamento detalhado do procedimento de busca foge aos interesses deste trabalho. Entretanto, o leitor interessado pode recorrer os textos [Mac92; CGL94; WVF97; CGL96] para um estudo mais aprofundado sobre o tema. Para efeito de entendimento deste trabalho, é bastante perceber que a técnica de Verificação Automática de Modelos realiza uma busca por exaustão no espaço de estados do modelo tentando satisfazer fórmulas escritas em lógica temporal, para isso recorre a representações mais compactas do espaço de estados como DDBOs por questões de eficiência.

Em se tratando do universo de modelagem em redes de Petri, a representação de alto nível do modelo é o próprio grafo de ocorrência, entretanto é necessária uma lógica temporal que possa expressar propriedades sobre o comportamento destes modelos. Em seguida, descreve-se minimamente a lógica temporal ramificada (*Computational Tree Logic* - CTL) e a sua variante que será usada para descrever fórmulas sobre o comportamento de modelos em redes de Petri Coloridas, o ASK-CTL.

3.2.1 Lógica Temporal

Apesar das redes de Petri serem uma ótima escolha de linguagem para a modelagem de sistemas de software complexos, as mesmas vantagens não se aplicam quando se deseja descrever propriedades sobre o comportamento dos mesmos. Neste caso, as lógicas e álgebras são mais indicadas por serem linguagens de especificação declarativas e não orientadas a modelos.

Lógica Temporal, tal como proposta por Pnueli [Pnu77], é um tipo especial de

Lógica Modal onde, além dos operadores da Lógica Proposicional, operadores temporais ou "modalidades" são acrescidos para que se possa descrever e verificar como os valores verdade das assertivas variam com o tempo [Sti89].

Ainda segundo proposto por Pnueli, Lógica Temporal é um sistema formal bastante útil à especificação e verificação de corretude de programas de computador de natureza reativa e concorrente [Pnu77]. Como exemplos de tais sistemas tem-se, sistemas operacionais e protocolos de comunicação.

As lógicas temporais são classificadas de acordo com a natureza do tempo (linear ou ramificado), a natureza de sua assertivas (proposicional ou de primeira ordem), a forma do tempo (discreto ou contínuo) e sentido temporal (tempo passado ou futuro). A este trabalho, só interessa o tipo proposicional, ramificado, de tempo discreto e com a linha de tempo no sentido futuro. Entretanto, é importante discutir, mesmo que superficialmente, o comportamento dos operadores das lógicas de tempo linear, como a LTL, para que se possa melhor compreender os operadores de uma lógica de tempo ramificado como o CTL, discutida na próxima seção.

Nos sistemas lógicos de tempo linear só há um futuro possível. Logo, os estados estão todos descritos ao longo de apenas uma linha de tempo. Nestes sistemas, os operadores temporais mais comuns são **G** ("sempre"), **F** ("em algum momento futuro"), **X** ("no próximo momento") e **U** ("até que").

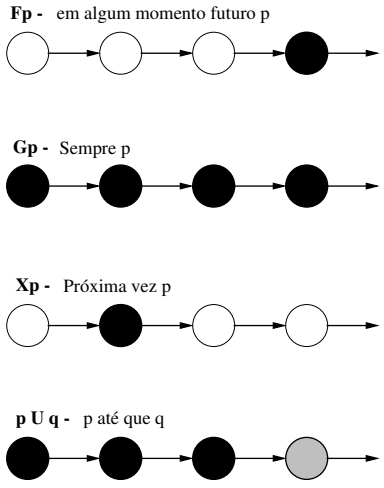


Figura 3.5: Visão intuitiva da semântica dos operadores temporais das LTL.

Na Figura 3.5 ilustra-se uma interpretação dos operadores de uma lógica temporal

linear utilizando diagramas de transição. Assim, na primeira ilustração, tem-se uma computação representada por uma seqüência de estados onde o último estado está diferenciado pelo círculo preto, indicando que a proposição **p** é verdade naquele estado. Isso significando que, **Fp** é verdade agora se em algum estado futuro a proposição **p** é avaliada verdadeira. Da mesma forma, pode-se intuir o comportamento dos demais operadores pelas ilustrações restantes.

3.2.2 Lógica Temporal Ramificada - CTL

A lógica temporal CTL é um sub-conjunto da lógica modal temporal ramificada definida por Clarke e Emerson [MA81], cujo acrônimo significa *Computation Tree Logic*. Os operadores de tempo de CTL ocorrem em pares da seguinte forma: quantificadores de caminho, **A** ("para todos os caminhos computacionais") ou **E** ("para alguns caminhos computacionais") precedendo os operadores de tempo linear **G** ("sempre"), **F** ("em algum momento futuro"), **X** ("próximo"), **U** ("até") e **V** ("a não ser que").

A interpretação desses operadores está ilustrada na Figura 3.6 de maneira intuitiva. Nas ilustrações, uma computação é representada por uma árvore de estados **M**, em que o primeiro círculo é o estado inicial s_0. Os círculos em preto representam os estados em que a proposição g é verdade. Assim, na ilustração que indica o comportamento do operador **EF**, o círculo em preto representa que para algum caminho, em um estado futuro a proposição g é verdade, avaliando a fórmula **EF**g para verdade agora. As ilustrações restantes podem ser interpretadas de forma similar.

As fórmulas em CTL são formadas pelos operadores temporais já comentados, proposições atômicas e conectores funcionais ($\wedge$, $\vee$, $\neg$, etc.), da seguinte forma:

1. Cada proposição atômica é uma fórmula CTL;

2. Se f e g são fórmulas CTL, então $\neg f$, $(f \wedge g)$ $\mathbf{AX}f$, $\mathbf{EX}f$, $\mathbf{A}(f\ \mathbf{U}\ g)$, $\mathbf{E}(f\ \mathbf{U}\ g)$ também são fórmulas.

Os demais operadores podem ser derivados a partir desses.

As fórmulas são avaliadas para falso ou verdadeiro de acordo com um modelo do comportamento do sistema chamado *modelo de Kripke*. Os detalhes do procedimento

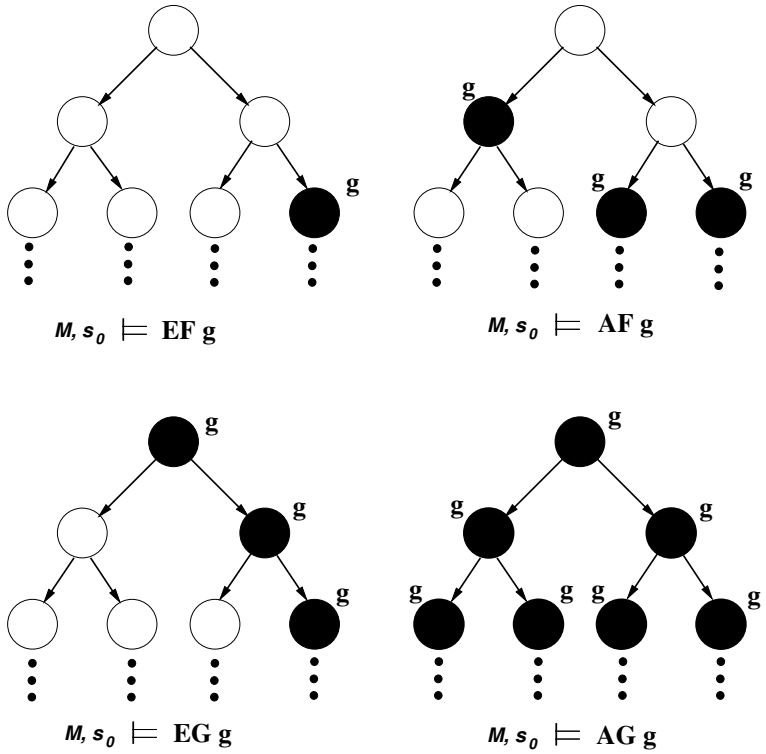


Figura 3.6: Visão intuitiva da semântica dos operadores CTL.

de verificação das fórmulas, da definição dos modelos de Kripke, bem como da semântica de CTL podem ser encontrados em [CGL96].

3.2.3 ASK-CTL

Normalmente, as propriedades de redes de Petri Coloridas são especificadas diretamente em termos de seus espaços de estados [Jen92]. O uso de uma lógica temporal construída especialmente para expressar propriedades sobre o espaço de estados de CPNs implica em ter-se uma maneira, bem conhecida e mais fácil de usar, de expressar um espectro bem maior de propriedades.

ASK-CTL é uma lógica similar a CTL que é interpretada sobre o espaço de estados de CPNs [CCM96]. De forma a possibilitar a verificação de propriedades eficientemente, o algoritmo de verificação é melhorado através do uso de grafos de componentes fortemente conectados (*Strongly Conected Components* - SCC's)[Jen97].

Os grafos de SCC's são um tipo especial de grafo derivados do espaço de estados, onde cada nó é um SCC. Cada SCC representa um sub-conjunto de nós no espaço de

estados, que têm a propriedade de cada nó ser alcançável a partir de qualquer outro nó deste sub-conjunto. Esses sub-conjuntos são mutuamente disjuntos, máximos e são partições do espaço de estados. Os grafos de SCC's são acíclicos e dois de seus elementos são ligados por um arco toda vez que houver um arco ligando dois nós, um em cada um dos dois SCC's.

Em ASK-CTL há duas categorias sintáticas de fórmulas definidas de forma mutuamente recursiva: fórmulas de estado e de transição que são interpretadas sobre o espaço de estados com relação à informações de estados e de transições. Essa extensão do CTL possibilita expressar um maior número de propriedades. Em termos de expressividade, as duas lógicas são equivalentes uma vez que os predicados escritos para fórmulas em ASK-CTL são limitados a proposições atômicas. As fórmulas de estado são definidas da seguinte forma:

- fórmulas de estado: $A ::= tt \mid \alpha \mid \neg A \mid A_1 \wedge A_2 \mid < B > \mid EU(A_1, A_2) \mid AU(A_1, A_2)$

onde tt significa *verdade*, α é uma função que mapeia marcações em valores booleanos e B é uma fórmula de transição. EU e AU são operadores do ASK-CTL formados pela combinação dos operadores de caminho E e A com o operador U (todos já comentados nas Seções 3.2.1 e 3.2.2).

As fórmulas de transição são, analogamente, definidas como segue:

- fórmulas de transição: $B ::= tt \mid \beta \mid \neg B \mid B_1 \wedge B_2 \mid < A > \mid EU(B_1, B_2) \mid AU(B_1, B_2)$

onde β é uma função que mapeia elementos de ligação em valores booleanos e A é uma fórmula de estado. O operador $< ... >$ torna possível a mudança entre fórmulas de estado e de transições.

Assim, o operador $EU(A_1, A_2)$ expressa a existência de um caminho computacional, iniciando a partir de uma certa marcação, em que A_1 é verdade até que se alcance uma marcação em que $A2$ seja verdade. Da mesma forma pode-se interpretar $AU(A_1, A_2)$, porém com a restrição de que a propriedade deve valer para todos os caminhos computacionais e não apenas um. A definição formal do ASK-CTL se encontra em [CCM96].

De maneira a facilitar a descrição de propriedades, algumas abreviações foram sugeridas pelos criadores do ASK-CTL. Para fórmulas de estado tem-se:

- POS $A \equiv EU(tt, A)$ – É possível alcançar um estado em que A seja verdade.
- INV $A \equiv \neg$POS $\neg A$ – A é verdade em todos os estados alcançáveis, ou A é verdade invariavelmente.
- EV $A \equiv AU(tt, A)$ – Para todos os caminhos, A seja verdade dentro de um número finito de passos, ou A certamente será verdade.
- ALONG $A \equiv \neg$EV$\neg A$ – Existe um caminho (infinito ou que termine em uma marcação morta) ao longo do qual A é verdade em todos os seus estados.
- $< B > A \equiv < B \wedge < A >>$ – Existe um estado imediatamente sucessor (M_1) no qual A é verdade e, B é verdade na transição entre o estado corrente e M_1.
- $EX(A) \equiv < tt > A$ – Existe um estado imediatamente sucessor em que A é verdade.
- $AX(A) \equiv \neg EX(\neg A)$ – A é verdade em todos os estados imediatamente sucessores, se houver algum.

As mesmas abreviações para fórmulas de transições também valem. Como o ASK-CTL foi implementado como uma biblioteca de funções em SML'97, estas abreviações são encontradas em forma de funções. A descrição da biblioteca ASK-CTL e como usa-la pode ser encontrada em [Uni96].

Para ilustrar o uso do ASK-CTL como linguagem para declaração de propriedades de redes de Petri Coloridas, recorre-se uma vez mais ao problema exemplo do jantar dos filósofos como descrito na Seção 3.1. Entretanto, o modelo CPN original ilustrado na Figura 3.3 será alterado para que o modelo apresente um espaço de estados finito, adequado à verificação automática de propriedades. Na Figura 3.7 ilustra-se o modelo CPN do jantar alterado com uma sessão de entrada no ambiente de jantar e um eventual envenenamento dos filósofos comensais. Ao modelo ilustrado na Figura 3.3, foram acrescentados o lugar **Fora**, que representa os filósofos fora da sala de jantar, e as transições **Senta a Mesa** e **Envenenado** que representam respectivamente a entrada dos filósofos no sistema e o envenenamento destes durante o jantar.

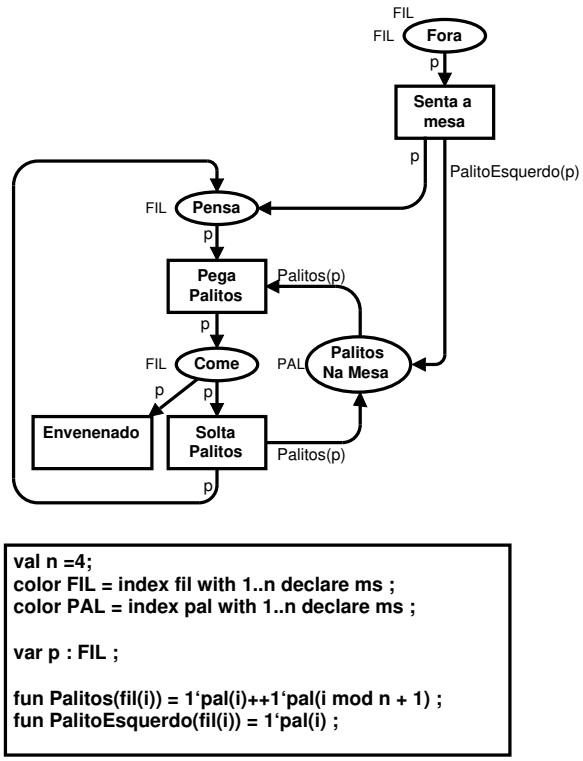


Figura 3.7: Modelo CPN da versão venenosa do jantar dos filósofos.

Como exemplo de uma possível propriedade a ser verificada sobre o espaço de estados do modelo ilustrado na Figura 3.7, tem-se :

```
fun EhEnvenenado n a = (Bind.System'Envenenado (1, {p= fil(n)}) = ArcToBE a);
val MinhaFormulaASKCTL = MODAL(POS(AF("fil(2) é envenenado",EhEnvenenado 2)));
eval_node MinhaFormulaASKCTL InitNode;
```

Evitando detalhar muito a sintaxe das sentenças acima, **EhEnvenenado** é um predicado que é avaliado como verdade caso, para o estado corrente, haja um elemento de ligação **fil(n)** no arco que leva a transição **Envenenado**. A fórmula **MinhaFormulaASKCTL** descreve em ASK-CTL a seguinte propriedade: *é possível que o filósofo 2 se envenene?*. A linha final é a forma como se avalia uma fórmula usando a sintaxe do ASK-CTL. O leitor interessado pode recorrer a [Uni96] para mais alguns exemplos de uso do ASK-CTL.

Capítulo 4

Solução de Reúso de Modelos em Redes de Petri Coloridas

Neste capítulo, detalha-se a solução de reúso introduzida no Capítulo 2. Para melhor entendimento, apresenta-se um cenário em que o reúso de modelos CPN é alcançado através da aplicação do método de armazenamento e da técnica de recuperação. Em seguida à descrição deste cenário, detalha-se o método e a técnica para, finalmente, contextualizar-se os esforços de reúso deste trabalho de acordo com as classificações apresentadas na Seção 2.1.

4.1 O Cenário de Reúso de Modelos

Para fins de entendimento, duas situações de modelagens são descritas neste cenário. Considere então, uma primeira situação em que parte de um sistema de software para controle de tráfego ferroviário é modelada. Um módulo deste sistema trata do controle das manobras que as máquinas devem efetuar para realizar a troca de vagões. Por exemplo, quando da chegada de um trem ao seu destino, há que se executar uma manobra para a troca de composições, onde o trem deixa a composição carregada para atrelar-se à composição vazia.

Neste caso, além das partes do modelo que representam o comportamento das estruturas dos trilhos e suas conexões, há as partes que controlam a troca de vagões das composições. Veja na Figura 4.1 uma ilustração das seções onde a troca de máquinas

das composições é efetuada.

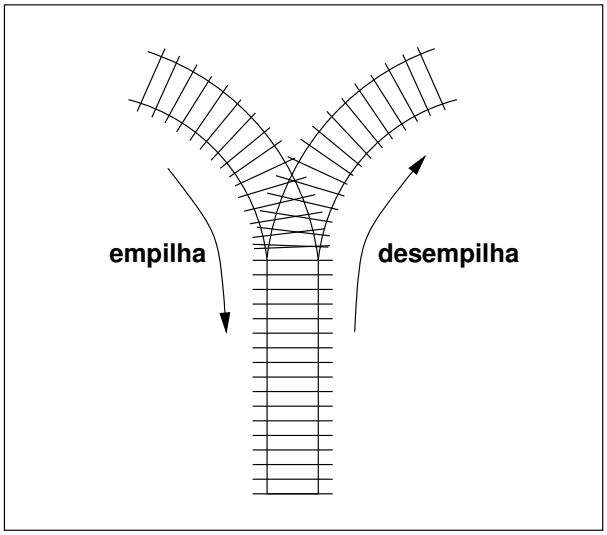


Figura 4.1: Junções do sistema de troca de composições de trens.

A situação ilustrada pela Figura 4.1 é um exemplo conhecido, onde o controle da troca de máquinas de acordo com as indicações da ilustração é modelado como o controle de uma estrutura de dados do tipo pilha [Knu68]. Não obstante, o mesmo sistema pode exigir o controle da troca de máquinas cuja modelagem seja o controle de uma estrutura do tipo fila, nos casos em que a composição seja manobrada em outro sentido na mesma malha. Os elementos manipulados por esta estrutura de dados podem ser de quaisquer tipos, neste contexto são de um tipo complexo que representa vagões. Entretanto, o mesmo modelo pode ser reaproveitado em diferentes contextos poupando esforços de modelagem futuros.

Nas Figuras 4.2 e 4.3, são apresentados os modelo CPN para controle de uma pilha e uma fila de inteiros respectivamente. Estes modelos são ótimos candidatos a serem "guardados" para uso futuro. A declaração de cores destes modelos pode ser facilmente alterada para utilizar uma cor mais complexa em representação aos elementos manipulados.

Considere agora que há um método com o qual se pode armazenar modelos para que o seu reúso seja sistemático. Assim, o projetista responsável pela modelagem do sistema de controle de trens pode identificar os modelos candidatos ao reúso e, seguindo os passos definidos no método, armazenar tais modelos de forma adequada, como por

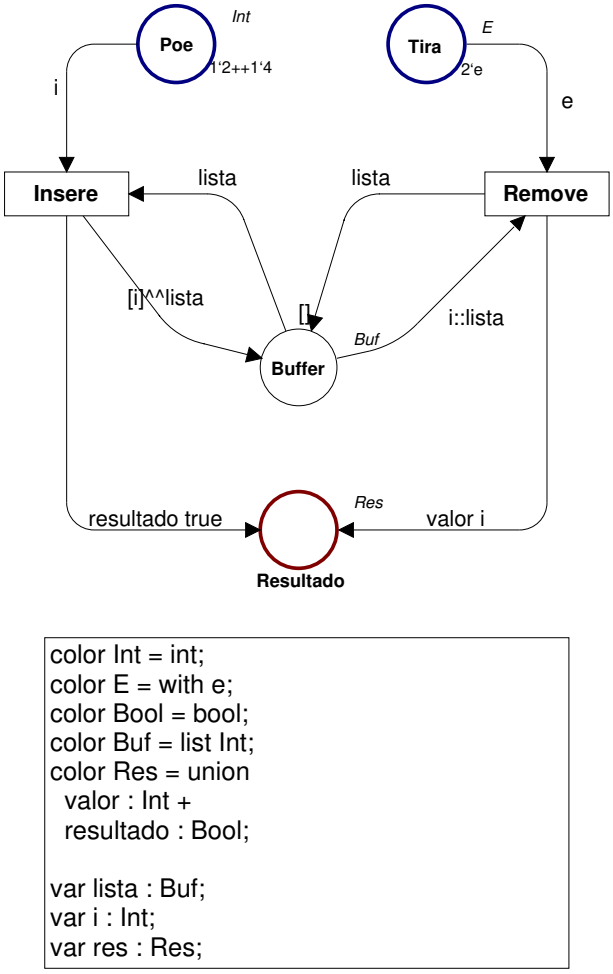
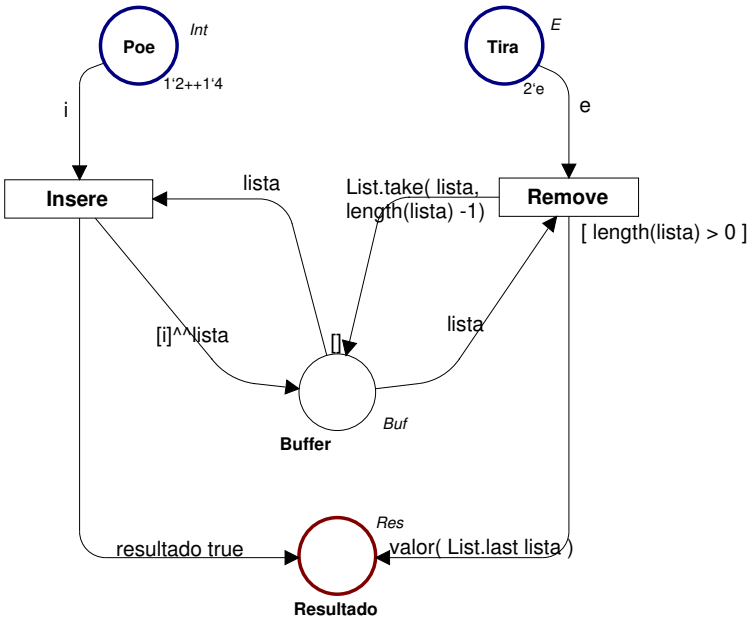


Figura 4.2: Modelo CPN de uma pilha de dados.

exemplo o modelo de controle da pilha (Figura 4.2) e o de controle da fila (Figura 4.3).

Em um outro momento, o mesmo projetista se encontra diante da tarefa de modelar formalmente a parte de um sistema flexível de manufatura que implementa o controle do empilhamento e desempilhamento de materiais nos depósitos de entrada e de saída das células de produção [BP99] como ilustrado na Figura 4.4. Neste sistema, ainda necessita-se de controle para colocação e retiradas de materiais de esteiras dos recursos de produção de cada célula de produção. Ou seja, ambos os procedimentos de controle de filas e pilhas são necessários nesta nova situação de modelagem, neste caso a pilha (ou fila) será de um tipo, estruturado ou não, que represente a disponibilidade de matéria-prima.

Partindo da constatação de similaridade, o projetista recorre ao repositório de modelos com o propósito de pesquisa-lo em busca de um modelo que apresente o compor-




```
color Int = int;
color E = with e;
color Bool = bool;
color Buf = list Int;
color Res = union
  valor : Int +
  resultado : Bool;

var lista : Buf;
var i : Int;
var res : Res;
```

Figura 4.3: Modelo CPN de uma fila de dados.

tamento desejado. Neste momento ele dispõe de uma técnica que lhe permite testar os modelos armazenados de forma automática retornando aqueles que correspondam ao critério de busca. O projetista pode economizar bastante tempo uma vez que lá se encontra um modelo que pode ser usado para controlar uma pilha de materiais com poucos ajustes.

Seguindo os passos indicados pela técnica de recuperação, o modelo armazenado durante o processo de modelagem do sistema de controle de tráfego ferroviário pode ser reusado no contexto de modelagem de um sistema flexível de manufatura. Após a conclusão deste novo processo de modelagem, é mais uma vez possível que o projetista identifique novos candidatos ao reúso incrementando e enriquecendo o repositório de modelos.

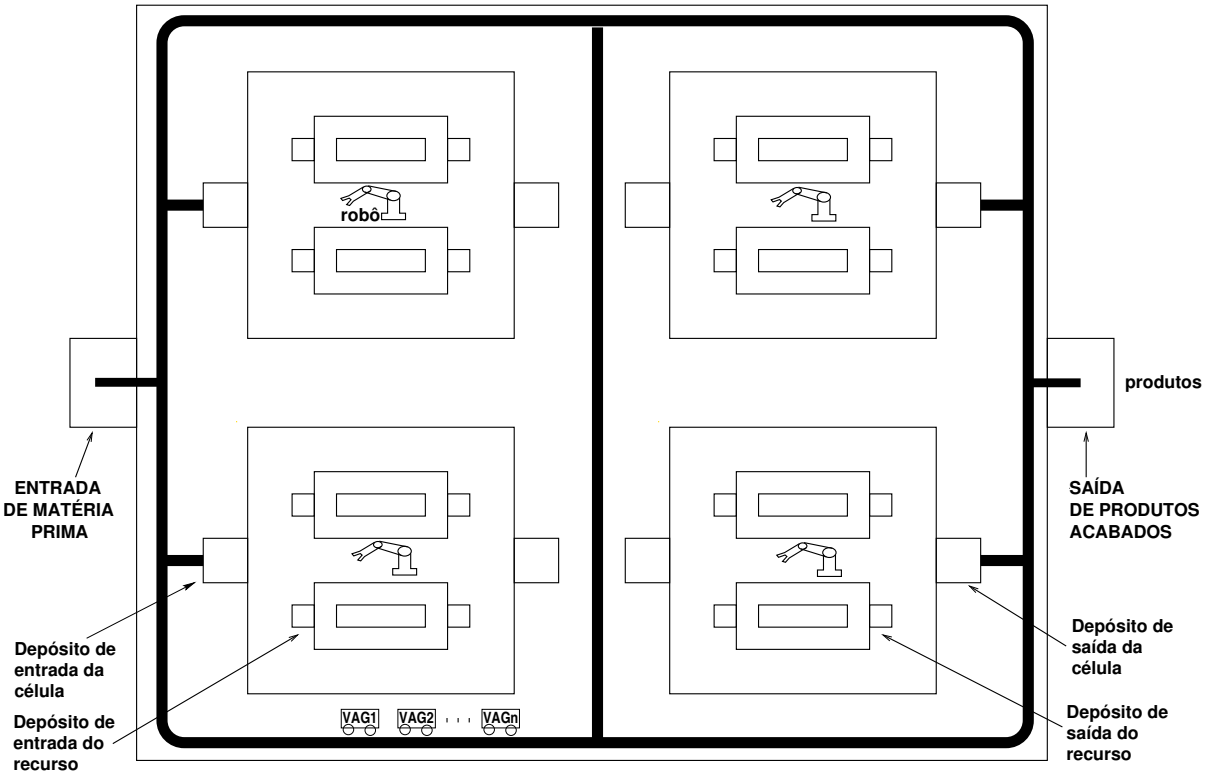


Figura 4.4: Sistema flexível de manufatura com 4 células.

No que segue, detalha-se o método introduzido neste trabalho. Para tanto, enumera-se os passos a serem seguidos para o armazenamento de modelos CPN em um repositório. Além deste detalhamento, ilustra-se a apresentação com um exemplo.

4.2 O Método de Armazenamento

O método introduzido neste trabalho baseia-se numa estrutura hierárquica de modelos CPN para implementar o repositório de modelos. É razoável supor que o conhecimento de modelagem a ser generalizado para uso posterior é bem compreendido pelo projetista que trabalha com Redes de Petri Coloridas Hierárquicas. O que se propõe então é que o projetista armazene os modelos que ele considera passíveis de reúso de acordo com uma classificação em domínios de aplicação.

Assim, decidiu-se por organizar o repositório em uma hierarquia de domínios, aos quais os modelos devem aderir ao serem armazenados. Por exemplo, um modelo CPN de um canal de comunicação pode ser armazenado sob um domínio chamado "Protoco-

los". De fato é o que ocorre com o repositório construído para ser utilizado como prova de conceito neste trabalho, há inicialmente dois domínios de aplicação sob os quais os modelos podem ser armazenados: estruturas de dados e protocolos de comunicação.

É útil salientar que a hierarquia comporta o incremento do número de domínios que se julgar necessário. Na realidade, como o repositório é, em última instância, um modelo de modelos, pode-se replicar sua estrutura fundamental separando ou mesmo distribuindo os domínios entre vários repositórios cada vez que se tornar necessário.

Assim, para que o repositório de modelos seja construído adequadamente, deve-se armazenar modelos seguindo os passos do método na seguinte ordem:

1. Define-se um modelo que será a estrutura fundamental do repositório considerando o uso de uma hierarquia de páginas com substituição de transições. A página CPN principal será o índice do repositório onde transições de substituição serão criadas para cada domínio de modelos.

2. Para cada novo modelo a ser armazenado deve-se criar uma nova página CPN. Essa página deve ser integrada ao repositório através de uma transição de substituição que será criada na página referente ao domínio a que o novo modelo pertence.

3. Se não houver um domínio adequado a classificação do novo modelo, deve-se criar uma nova página representando o primeiro nível hierárquico do novo domínio e uma transição de substituição para este na página de índice. Na página do novo domínio deve-se criar um modelo que será o ambiente que usa os modelos inseridos naquele domínio. Este primeiro ambiente será padrão para todos os outros modelos armazenados nas próximas sessões de uso do método.

4. Caso o modelo sendo armazenado possa ser classificado sob algum domínio existente, deve-se repetir a mesma estrutura hierárquica criada para utilizar os modelos existentes sob este domínio de forma que o novo modelo possa ter o mesmo contexto de uso dos modelos já armazenados.

5. As declarações de cores e variáveis relativas ao modelo devem ser adicionadas ao nó de declaração global, onde estão todas as cores e variáveis dos domínios

armazenados no repositório. Recomenda-se que ao fazê-lo, o projetista tenha o cuidado de separar as declarações específicas de cada modelo (ou domínio) através de comentários.

6. Os nomes dos lugares e transições que representam a interface do modelo, devem ser padronizados pelo projetista no momento do armazenamento no repositório. Ao final dos nomes de lugares e transições de cada modelo será adicionado o nome da página CPN que o contém. Esse passo se aplica aos modelos de um mesmo domínio para que as propriedades sobre o comportamento destes modelos possam ser parametrizadas, permitindo com isso que essas sejam avaliadas nos diversos modelos do domínio. Os nomes desses lugares e suas cores devem ser replicados em uma caixa de texto dentro da página daquele domínio.

7. Para cada domínio de modelos há uma lista contendo os nomes das páginas que contém os seus modelos. Quando da inserção de um novo modelo, deve-se obrigatoriamente inserir mais um ítem na lista do domínio em questão com o nome da página que o contém.

8. Quando a página do novo modelo for ligada à hierarquia através de uma transição de substituição na página do seu domínio, deve-se incrementar a cor que representa os ítens desse domínio. Esse passo é muito importante para que técnica de recuperação funcione de forma automática.

9. As cores e variáveis criadas para cada domínio de modelos devem ser replicadas em caixas de texto dentro de cada página daquele domínio. Os tipos de dados devem ser substituídos por mnemônicos do tipo "X_<nome_da_cor>". Isto torna possível uma integração automática do modelo recuperado em um novo projeto através da substituição do mnemônico pelo tipo de dados específico do modelo no novo projeto.

10. Como última medida, deve-se realizar uma verificação sintática em todo o repositório para garantir que não haja problemas de conflitos de nomes, evitando assim possíveis falhas do procedimento automático de busca por problemas de sintaxe introduzidos durante processo de armazenamento.

Estes são os passos a serem seguidos para que o repositório de modelos CPN esteja pronto para uso quando for necessário recuperar algum modelo. Acompanhe no que segue uma sessão de uso do método em que o modelo de controle de uma pilha de dados, identificado na primeira situação descrita para o cenário introduzido na Seção 4.1, é armazenado em um repositório de modelos.

4.3 Construindo o Repositório e Armazenando Modelos

O armazenamento do modelo identificado na primeira situação de modelagem descrita pelo cenário de reúso do começo deste capítulo, é dividido em quatro etapas que são ilustradas pelo esquema de uso do método exibido na Figura 4.5.

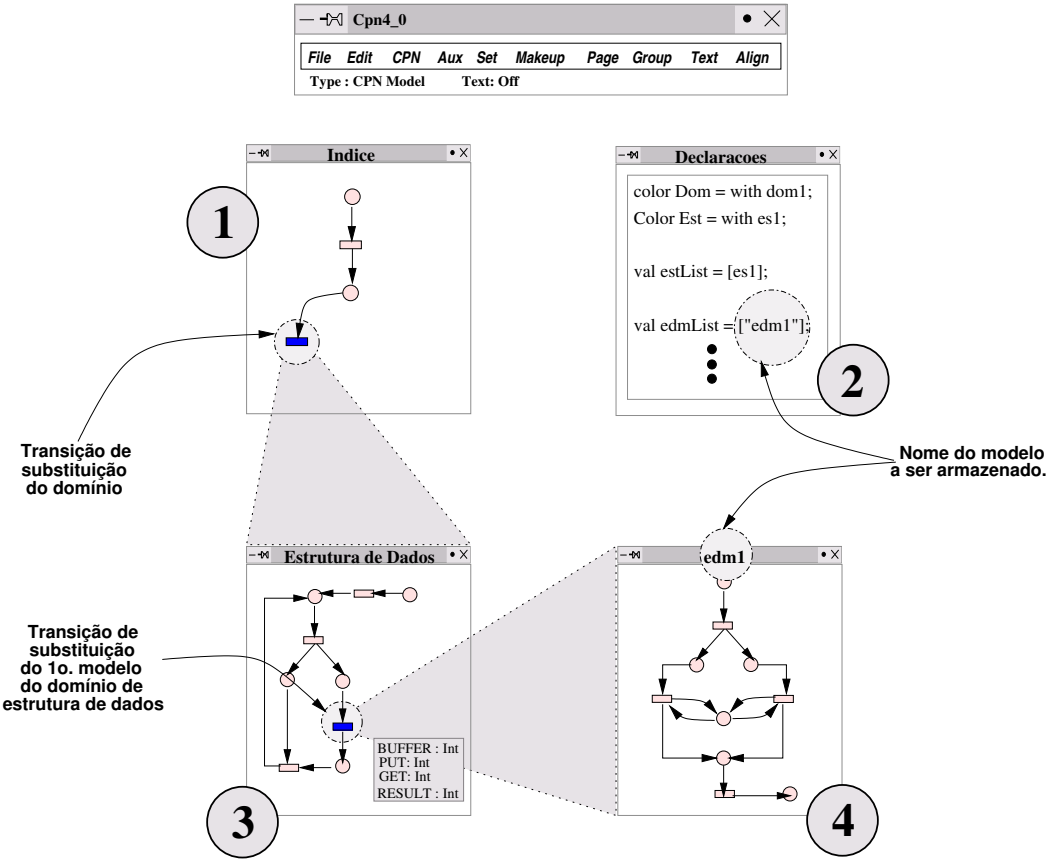


Figura 4.5: Esquema ilustrativo das etapas necessárias a criação do repositório.

A princípio, o modelo do repositório ainda não está construído, isso é feito na

primeira etapa ilustrada na Figura 4.5. Executa-se o Design/CPN e cria-se um novo modelo através do ítem **New** do menu **File** e uma primeira página é exibida para edição, onde deve ser criado o índice do repositório. Esta página, inicialmente, conterá dois lugares que comportarão fichas que determinam que domínio está sendo tratado.

Na página **Indice** recém-criada, haverá apenas uma transição de substituição que representa a página CPN do domínio de estruturas de dados. Na Figura 4.6 mostra-se a página CPN criada após a inserção de dois domínios de aplicação: o de estruturas de dados representado pela transição de substituição **EstDeDados**, que inclui os modelos trabalhados nesta sessão de uso, e o de protocolos representado pela transição de substituição **Protocolos**. Nesta figura, podem ser identificadas mais duas transições que não representam domínio algum, são elas **slot1** e **slot2** que foram criadas apenas como sugestão de ponto de partida para criação de outros domínios. O lugar **selecao** apenas serve para armazenar uma ficha sem significado especial (ficha de cor **E**) que habilita a transição **sel** possibilitando o início do processo de seleção do domínio. A função do lugar **Inicio** será discutida mais adiante.

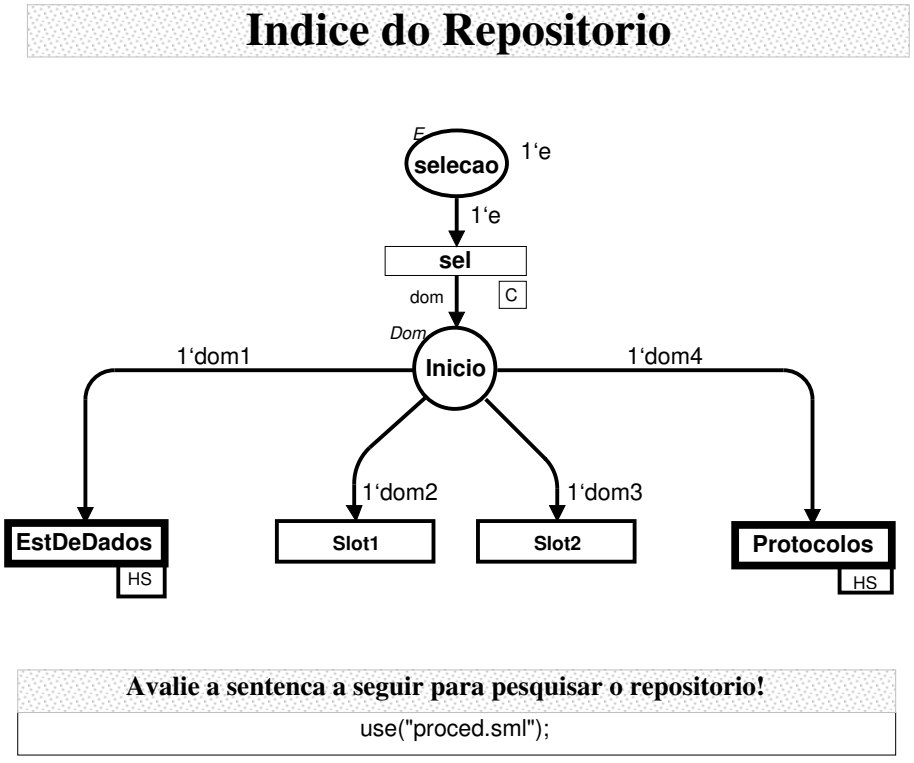


Figura 4.6: Modelo CPN do índice do repositório.

Após a criação da página índice do repositório como ilustrado na Figura 4.5, inicia-se a segunda etapa, que inclui a criação do nó de declarações globais. Nesta etapa, as cores dos lugares da página **Indice** são declaradas e os domínios são descriminados por uma cor que é formada por um tipo enumerado, por exemplo: **dom1** para o primeiro domínio. A cada novo domínio, repete-se a cadeia de caracteres **dom**X incrementando-se o número do seu final. Isto permite a parametrização dos domínios para verificação automática das sentenças no procedimento de busca, como será discutido na Seção 4.6.

Duas listas são criadas para cada domínio: uma lista de nomes de variáveis (**estList**) que serão referenciadas dinamicamente no processo de busca e uma lista de nomes das páginas (**edmPagList**) contendo os modelos armazenados naquele domínio. Essas listas estão representadas na ilustração dentro do nó de declarações.

No repositório criado e usado para teste, o nó de declarações globais foi colocado numa página separada. Na Figura 4.7 mostra-se o nó de declarações tal qual ficou após a construção do repositório. As listas criadas para o domínio de estruturas de dados foram nomeadas como **estList** e **edmPagList**, respectivamente. A variável **doms** de referência para uma cor do tipo **Dom**, foi criada para armazenar a resposta do projetista (obtida por uma caixa de diálogo) quando da escolha do domínio. Essa variável é alterada por um código SML'97 criado para ser avaliado durante a ocorrência da transição **sel**, após o qual uma ficha representando o domínio selecionado é depositada no lugar **Inicio**, habilitando apenas a transição de substituição correspondente. Por exemplo, caso o projetista selecione o domínio de estruturas de dados, uma ficha de valor **1'dom** será depositada no lugar **Inicio** e a transição **EstDeDados** estará habilitada.

A terceira etapa da sessão de uso ilustrada na Figura 4.5 corresponde à criação da página de domínio, segundo nível da hierarquia para os modelos desse domínio. Nesta página, um modelo que serve como ambiente de uso dos modelos do domínio deve ser construído. Neste ambiente é definida uma marcação inicial para o modelo, de forma que o espaço de estados gerado através dela descreva o comportamento necessário para verificação de propriedades.

Veja como exemplo o modelo da pilha de dados. Quando armazenado, este modelo manipula inteiros; e para que se verifique o funcionamento correto de uma pilha, apenas dois inteiros são necessários. Na Figura 4.8 exibe-se o modelo do domínio de estruturas

No de Declaracoes

```
(*---- Definicao das cores para os dominios ----*)
color Dom = with dom1 | dom2 | dom3 | dom4;
color Est = with es1 | es2;
color Pro = with pr1 | pr2;

val doms = ref dom1;
var dom : Dom;
val proList = [pr1,pr2];
val estList = [es1,es2];
val prmPagList = ["prm1","prm2"];
val edmPagList =["edm1","edm2"];
var pro : Pro;
val es = ref es1;
val pros = ref pr1;
var est : Est;

(* ---- Declaracoes  do dominio de estruturas de dados ----*)
color Buf = list Int;

color Res = union
  valor  : Int +
  resultado : Bool;

var lista : Buf;
var i : Int;
var res : Res;
```

Figura 4.7: Nó de declarações global do repositório.

de dados onde, o ambiente de uso dos modelos da pilha e da fila estão construídos. Neste caso, quando o domínio de estrutura de dados é selecionado, uma ficha **1‘dom1** é depositada no lugar **Inicio** habilitando a transição **EstDados**.

Seqüencialmente, o valor da variável de referência **es** é modificado para que apenas o espaço de estados de um dos modelos armazenados seja gerado. Assim, primeiramente a variável **es** assumirá o valor **es1**, causando o depósito de uma ficha de valor **1‘es1** no lugar **domest**, após o disparo da transição **EstDados**. Automaticamente, o espaço de estados do modelo **edm1** é gerado com a marcação inicial representada pela lista **1‘[2,4]**, que é depositada no lugar de entrada **InEsm1** do seu ambiente de uso após o disparo da transição **est1**. Em seguida, as propriedades podem ser verificadas e, após o resultado da verificação ser publicado ao projetista, o mesmo processo se repete para os próximos modelos armazenados de forma similar até que não haja mais nenhum.

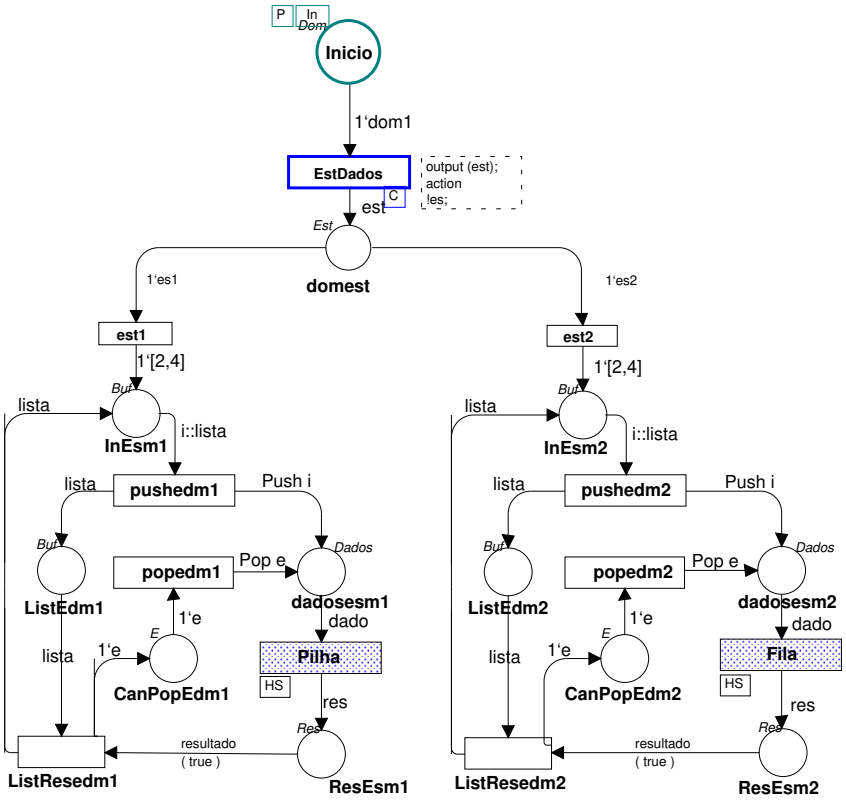


Figura 4.8: Modelo do domínio de estruturas de dados.

É importante ressaltar que, uma vez recuperado o modelo, este deve ser utilizado como definido pelo ambiente de uso (parte hachurada da Figura 4.8). Observe que, caso o projetista não use o modelo desta forma, não há garantias de que o modelo se comporte como especificado. Este ambiente de uso, no caso dos modelos do domínio de estruturas de dados, foi construído de forma que, por exemplo, o modelo da pilha só tivesse ativada sua operação de desempilhamento quando já houvesse elementos na pilha. Observe que as operações de empilhamento e desempilhamento são ativadas, respectivamente, pelo disparo das transições **pushedm1** e **popedm1**, que depositam um inteiro a ser empilhado ou uma ficha sem valor (**e**) no lugar **dadosesm1**.

Uma vez criado o ambiente de uso, deve-se criar uma transição de substituição para a página contendo o modelo armazenado, por exemplo, na Figura 4.8, a transição **Pilha** representa o modelo da pilha de dados. Nesta página, as ligações com a transição de substituição são feitas, as cores das inscrições são substituídas pelas criadas no nó de declarações e as informações de nomes dos lugares e cores que correspondem a interface do modelo são copiadas para uma caixa de texto dentro da página do domínio. Observe

que os nomes dos lugares e transições que compõem a interface do modelo devem ser complementados com o nome da página CPN que os contém. Veja na Figura 4.9, por exemplo, como ficou o modelo CPN da pilha após armazenado no repositório.

No modelo ilustrado na Figura 4.9, o depósito de fichas nos lugares **PUTedm1** e **GETedm1** representa, respectivamente, a ativação das operações de empilhamento e desempilhamento de inteiros. As transições **pushedm1** e **popedm1** têm a função de retirar, em momentos distintos, fichas dos tipos **Int** e **E** do lugar **dadosP** e deposita-las respectivamente nos lugares **PUTedm1** e **GETedm1**, ativando as operações correspondentes. A aparentemente inútil redundância dos lugares **RESULTedm1** e **Retornoedm1** tem uma razão bem forte de ser: caso este lugar fosse *porta* e *socket* da transição de substituição **Pilha**, não seria possível verificar propriedades cujas proposições incluíssem o seu nome como integrante da página onde o modelo estivesse armazenado; este lugar pertenceria à página do domínio.

Por último, e não menos importante, a sintaxe de todo o modelo do repositório deve ser verificada, concluindo a quarta e última etapa da ilustração. Caso a verificação sintática não acuse algo errado, o repositório está criado e pronto para uso.

4.4 A Técnica de Recuperação

A técnica de recuperação de modelos desenvolvida neste trabalho consiste em identificar os modelos que atendem a uma especificação escrita na lógica temporal ASK-CTL. Esta lógica foi desenvolvida especialmente para descrever propriedades sobre o comportamento de modelos CPN em termos de proposições sob o seu espaço de estados. A identificação é feita através da aplicação da técnica de Verificação Automática de Modelos, introduzida no Capítulo 3.

Assim como feito na descrição do método de armazenamento, a técnica de recuperação é apresentada através da enumeração de seus passos na seguinte ordem:

1. Identifica-se que parte do modelo em desenvolvimento pode ser encontrada já modelada no repositório.

2. Recorre-se ao repositório de modelos e identifica-se o domínio de modelos que

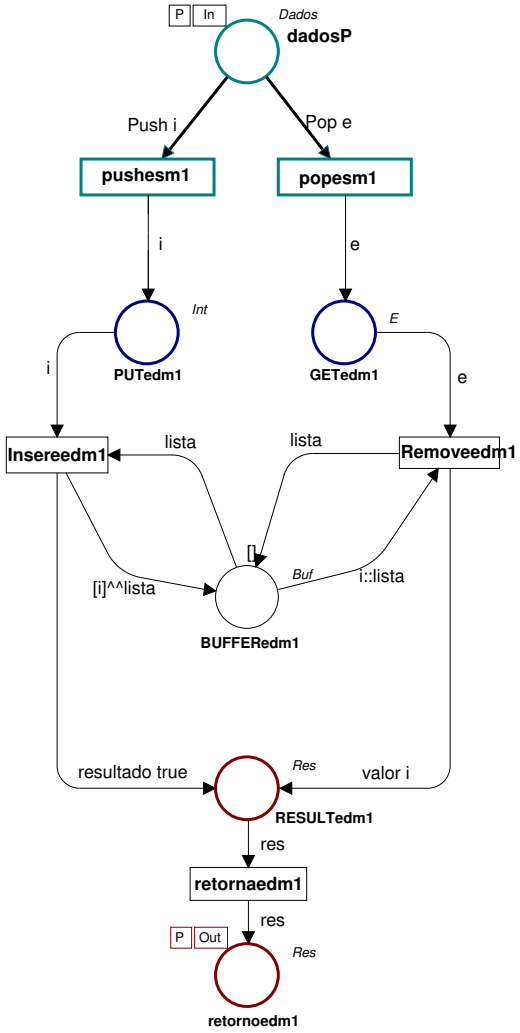


Figura 4.9: Modelo CPN da pilha armazenada no repositório.

será alvo da busca. Essa seleção prévia de domínio pode levar a uma redução do espaço de estados, reduzindo o tempo na busca por modelos.

3. Na página relativa ao domínio escolhido há informações pertinentes aos lugares e transições que formam a interface padronizada dos modelos armazenados naquele domínio. Com essas informações, escreve-se a sentença de busca em ASK-CTL expressando um comportamento desejado no modelo a ser recuperado e grava-se esta sentença em um arquivo.

4. Na página **Indice** do repositório, seleciona-se e avalia-se o código SML'97 que ali se encontra para iniciar o procedimento interativo de busca. O projetista será requisitado a informar o domínio de busca e o arquivo contendo a sentença de

busca.

5. Automaticamente, o procedimento verifica cada modelo armazenado sob o domínio indicado e, caso um ou mais destes seja modelo para a especificação da sentença de busca, o projetista será requisitado a informar o nome do arquivo em que será gravado o modelo recuperado. O procedimento não para até que se tenha verificado todos os modelos do domínio, isso mesmo que já se tenha encontrado um candidato a recuperação, portanto pode-se ter múltiplos resultados para uma única pesquisa.

A seguir, acompanhe passo-a-passo uma sessão de uso da técnica de recuperação onde o modelo procurado no repositório é a pilha de dados identificada e armazenada na primeira situação descrita pelo cenário de reúso.

4.5 Usando a Técnica de Recuperação

Considere que o repositório de modelos está devidamente populado e sua construção foi feita de acordo com o método de armazenamento. A sessão de busca a seguir é ilustrada pela Figura 4.10 e está dividida em cinco etapas.

A primeira etapa é carregar o modelo CPN do repositório de modelos e prepara-lo para uso. Tal preparação consiste em, após carregado o modelo, ativar a ferramenta de manipulação do grafo de ocorrência através do ítem **Enter Occ Graph** do menu **File**. Nesta etapa todo o código SML'97 para manipulação dos grafos de ocorrência do Design/CPN é carregado e uma verificação sintática do repositório é feita.

Na segunda etapa de acordo com a ilustração, identifica-se que domínio deve ser investigado. Em seguida, guiando-se pelas informações de nomes contidas na caixa de texto da página do domínio, deve-se escrever uma especificação de parte do comportamento do modelo a ser recuperado.

Essa especificação é a sentença de busca do modelo e deve ser escrita em dois passos: primeiro as proposições atômicas − que são predicados escritos em SML'97 usando os nomes de cores, de lugares e transições parametrizados pelo nome da página − e então escreve-se a fórmula em ASK-CTL que expressa um comportamento do modelo em

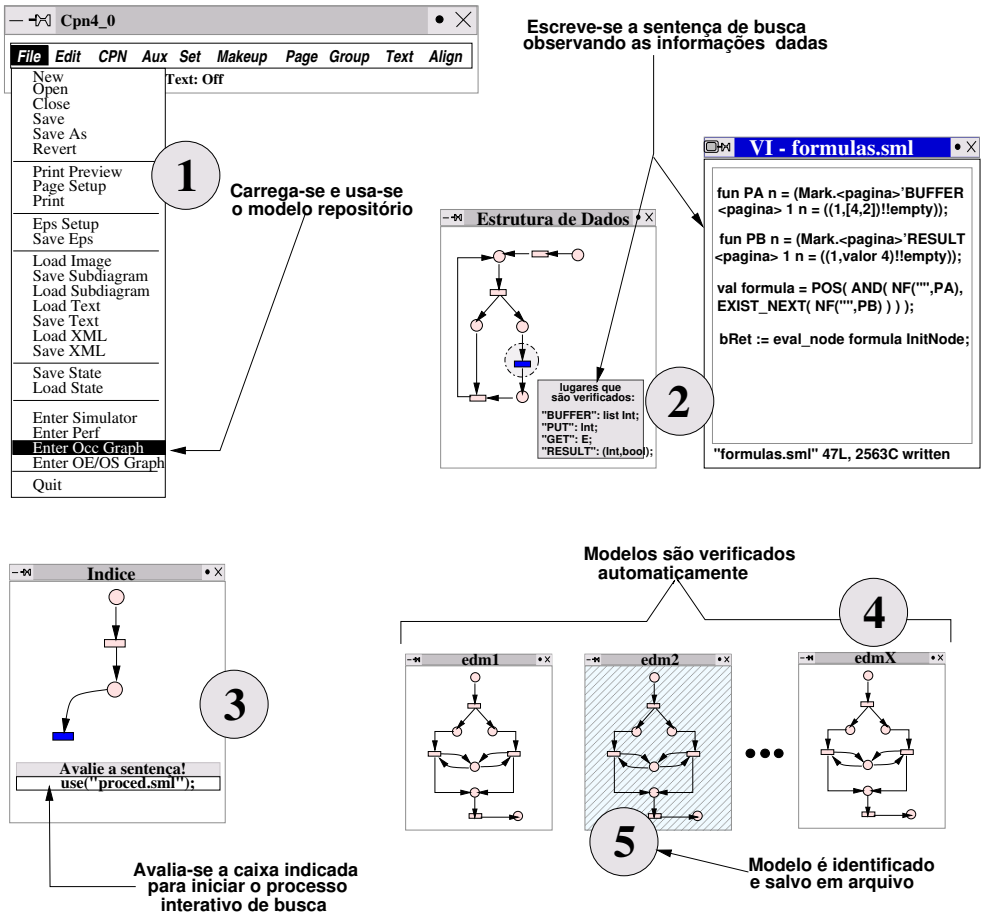


Figura 4.10: Esquema ilustrativo de uma sessão de uso da técnica de recuperação.

termos dessas proposições atômicas. Veja a seguir, um exemplo de sentença de busca escrita para o contexto de busca pelo modelo da pilha de dados no cenário de reúso:

```
fun PA n = (Mark.<pagina>'BUFFER<pagina> 1 n = ((1,[4,2])!!empty));
fun PB n = (Mark.<pagina>'RESULT<pagina> 1 n = ( (1,valor 4)!!empty ) );
val formula = POS(AND(NF("buffer",PA),EXIST_NEXT(NF("result",PB))));
```

Em síntese, a sentença de busca acima descreve parte do comportamento de uma pilha de inteiros (o comportamento que basta ao projetista para recuperar o modelo desejado neste contexto). A proposição **PA** descreve a situação em que o modelo contém dois inteiros armazenados em seu *buffer* interno, 2 e 4 nesta ordem. A proposição **PB** informa que há 4 no lugar de saída do modelo como resultado de uma operação de retirada de um elemento do *buffer* interno.

A fórmula em ASK-CTL descreve o seguinte comportamento: se o estado em que

PA é verdade for alcançado, então no próximo estado **PB** será verdade. Ou seja, caso tenha sido realizada uma operação de inserção de um 2 e depois um 4 no modelo e uma operação de retirada de elemento seja realizada, o inteiro resultado desta operação será o 4. Essa propriedade é característica singular de uma estrutura de dados do tipo pilha.

Na terceira etapa, deve-se avaliar o código da caixa de texto que se encontra na página de índice do repositório. Para faze-lo, deve-se selecionar a caixa de texto e pressionar a combinação de teclas **ALT** + ;, isso iniciará o procedimento interativo de busca. Logo em seguida, o projetista será requisitado através de janelas de diálogo a identificar o domínio alvo e o arquivo que contém a sentença de busca.

A Figura 4.11 contém a página de hierarquia do repositório obtida ao final do processo de criação seguindo os passos do método. É através desta página que o projetista deve escolher o domínio de aplicação da busca.

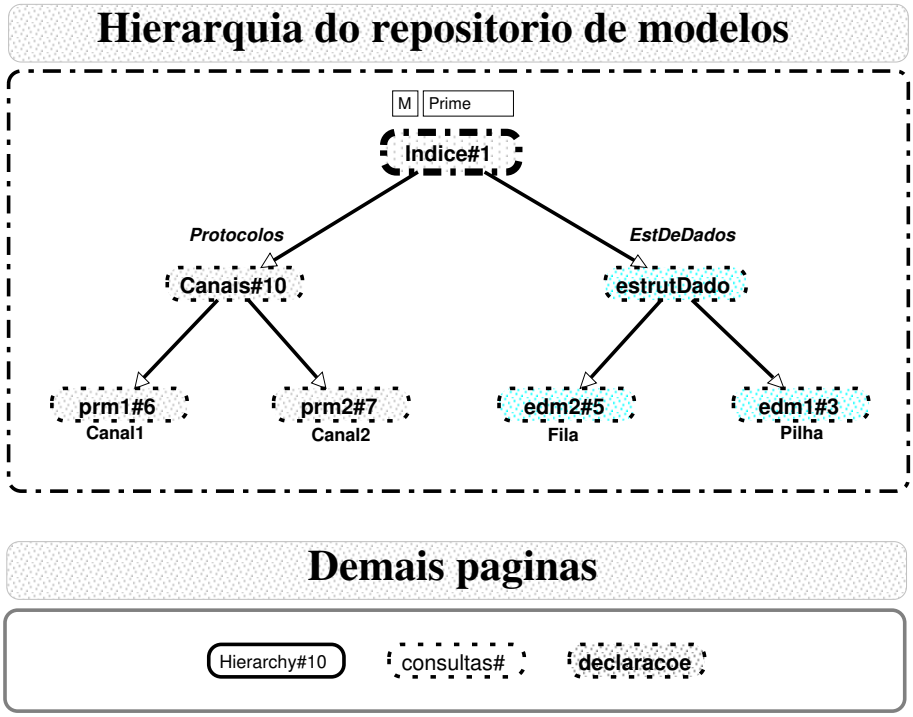


Figura 4.11: Página de hierarquia do repositório de modelos.

O restante do processo de busca é automático. As etapas quatro e cinco ilustradas pelo esquema apresentado na Figura 4.10 ocorrem simultaneamente, uma vez que cada modelo sendo investigado pode ser um candidato identificado. Isso implica em uma

interrupção a cada momento em que um modelo é identificado como candidato a recuperação, requisitando ao projetista o nome do arquivo em que o modelo recuperado será gravado.

Uma vez concluída a busca com sucesso, o projetista tem em seu poder um ou mais modelos que correspondem àquela especificação escrita na sentença de busca. A próxima tarefa é integrar o modelo (ou um dos modelos) recuperado(s) ao projeto em desenvolvimento.

4.6 Observações Adicionais

O cerne da técnica de recuperação de modelos CPN é o procedimento de verificação automática das propriedades ASK-CTL contra os modelos CPN. Este procedimento é um código SML'97 que percorre todos os modelos de um domínio mudando dinamicamente a marcação inicial do repositório de forma a gerar apenas o espaço de estados (grafo de ocorrência) referente ao modelo hora investigado, como já foi comentado nesta mesma seção.

Este procedimento de alteração da marcação inicial para cada modelo, é bastante eficiente mas não resolve o problema de explosão do espaço de estados para todos os casos. É fundamental que o usuário do repositório que esteja armazenando um modelo seja criterioso na escolha da marcação inicial que deve colocar como sugestão de contexto de uso inicial. Veja como exemplo o caso dos modelos que controlam as estruturas de dados pilhas e filas, a escolha de uma marcação inicial com apenas dois inteiros é suficiente para a verificação de quaisquer propriedades destes dois modelos, resultando em um espaço de poucas dezenas de estados. Entretanto, se a escolha para marcação inicial fosse a de se ter mais de quatro inteiros, o tamanho do espaço de estados atingiria facilmente algumas dezenas de milhares de estados.

A implementação da busca automática consiste em realizar um pré-processamento da sentença de busca para identificar os nomes delimitados pelos símbolos "<" e ">" que parametrizam as proposições e substituí-los pelos nomes das páginas que contém os modelos do domínio sendo investigado. Isso, e a avaliação da sentença de busca são feitos para cada modelo do domínio. Caso a avaliação da sentença de busca seja

verdadeira, o modelo deve ser salvo em um novo arquivo para posterior reúso. Caso a sentença seja avaliada como falsa, uma mensagem é apresentada notificando a falha.

A seguir, a abordagem de reúso elaborada neste trabalho é enquadrada de acordo com as classificações descritas no Capítulo 2.

4.7 Categorização do Trabalho de Reúso

É importante situar o contexto de pesquisa no qual se enquadra este trabalho de acordo com os levantamentos mais aceitos sobre a área de reúso de software. Neste sentido, classifica-se a abordagem de reúso de modelos CPN com relação aos dois trabalhos comentados no Capítulo 2.

Assim, neste trabalho a idéia de reúso de software na etapa de modelagem formal de sistemas de software, onde os artefatos de software objetos de reúso são modelos formais, com semântica precisa, é introduzida. De acordo com a classificação de Krueger, tais artefatos se enquadram como "esquemas de software" e isso posiciona este trabalho na quarta categoria dessa classificação. Quanto à taxonomia tem-se a seguinte análise:

- **Abstração:** Modelos CPN são os artefatos a serem reusados e suas descrições são feitas em uma linguagem de especificação orientada a propriedades (Lógica Temporal).

- **Seleção:** Uma técnica de recuperação baseada em Verificação Automática de Modelos é definida utilizando os recursos da ferramenta Design/CPN. Essa técnica permite a seleção e recuperação de modelos CPN armazenados em um repositório.

- **Especialização:** Os modelos serão adaptados pelo projetista de forma manual. Neste aspecto, embora fora do escopo de objetivos deste trabalho, é importante citar a possibilidade de automatização do processo de adaptação de modelos CPN, pesquisa já está em desenvolvimento [APKCG00].

- **Integração:** Os modelos recuperados são integrados aos novos projetos através de mecanismos de fusão de lugares e substituição de transições fornecidos pelo próprio formalismo de Redes de Petri Coloridas Hierárquicas. Este procedimento também é passível de automatização.

Considerando o segundo trabalho de classificação de abordagens na área de reúso de software, tem-se o espaço de cinco dimensões idealizado por Dusink [Dus92]. Este trabalho situa-se no primeiro eixo como sendo uma abordagem composicional, entretanto, pretende-se, no segundo eixo, o reúso de modelos de forma direta, da forma como foram recuperados (reúso caixa-preta), bem como o reúso de modelos que possam ser adaptados (reúso caixa-branca), uma vez que o processo de adaptação automática de modelos pode ser incluído na abordagem.

Continuando, no terceiro eixo dimensional proposto por Dusink, considerando o nível de abstração tratado, que define que artefato é alvo de reúso, trata-se de modelos formais. Quanto ao quarto eixo, o foco deste trabalho é em produtos, em tratar os modelos e não os processos de modelagem. Finalmente, no quinto eixo, trata-se de aspectos relativos tanto a construção de modelos reusáveis quanto à aplicação de reúso de modelos, muito embora a ênfase tenha sido na aplicação de modelos reusáveis.

4.8 Sumário

Neste capítulo foi apresentada uma aplicação dos princípios de reúso de software às atividades de modelagem formal de software em Redes de Petri Coloridas Hierárquicas. Esta aplicação resultou no desenvolvimento de um método de armazenamento e uma técnica de recuperação de modelos CPN cujos exemplos de uso foram ilustrados através de um cenário de reúso descrito na Seção 4.1. No capítulo seguinte são feitas as observações finais e algumas sugestões para evolução da presente pesquisa em reúso de modelos são propostas como trabalhos futuros.

Capítulo 5

Conclusão

Neste capítulo, os trabalhos relacionados à pesquisa de reúso de software relatada nesta dissertação são sumarizados e comentados. Os resultados obtidos, quais sejam, a definição e implementação do método e da técnica para promover o reúso de modelos CPN, são comparados com os objetivos declarados no Capítulo 1. Além disso, discute-se os possíveis desdobramentos desta pesquisa em trabalhos futuros.

5.1 Trabalhos relacionados

Ao longo da pesquisa e revisão bibliográfica sobre o cruzamento dos temas reúso de software e métodos formais, vários trabalhos abordam o tema de reúso de software através do tratamento de artefatos do tipo código fonte ou objeto sob o termo "componentes de software". A relação entre componentes de software e métodos formais é feita pelo uso de linguagens formais como notação matemática útil para o processo de recuperação automática de componentes.

Assim, tem-se os trabalhos da Jeanete Wing e Amy M. Zaremski [ZW95b; ZW95a] que buscam promover o reúso de código através de mecanismos de recuperação de componentes de software orientados a especificação. Nestes trabalhos, os componentes de software possuem descrições comportamentais escritas em *Larch*[1] e podem ser recuperados de seus repositórios através de técnicas como a prova de teoremas como já foi citado anteriormente na Seção 2.2. Outro trabalho interessante é do Oscar

[1] Larch é uma abordagem para especificação formal de módulos de programas [ZW95b].

Nierstrasz [NGT92] que pesquisa diferentes relações de substituição entre componentes de software baseadas em protocolos de interação entre componentes. Tais relações são utilizadas para determinar se um componente de software pode ser utilizado em um contexto em que o seu substituto funcionava.

Ambos os trabalhos supracitados usam métodos formais para promover reúso de software na fase de codificação, as atividades de reúso são basicamente recuperação de código assistida (ou auxiliada) por métodos e técnicas formais. Este trabalho tem os mesmo objetivos mas voltados para a atividade de modelagem de sistemas. O uso de prova de teoremas nos dois trabalhos se dá pelo fato de que ambos apenas dispõem de especificações de propriedades tanto na pesquisa quanto na anotação do componente de software, o artefato não possui uma representação do seu comportamento que possa ser verificada automaticamente. Neste trabalho, as propriedades estão todas codificadas no espaço de estados do modelo, restando assim saber percorre-lo para provar a validade ou não de uma propriedade desejada.

No sentido de reusar os esforços de modelagem (ou de especificação) anteriores, os únicos trabalhos relacionados que trilham caminhos similares aos deste trabalho são o do Matthew B. Dwyer [DAC99] sobre padrões de especificação de propriedades e o do Martin Naedele e Jorn W. Janneck [NJ99] sobre padrões de projeto em Redes de Petri, este último contribuindo diretamente no reúso de soluções de modelagem para problemas recorrentes em decisões de projeto em Redes de Petri. Ambos os trabalhos têm o objetivo de contribuir com o reúso sistemático de artefatos de software mais abstratos (especificações e modelos), valendo citar que o primeiro já dispõe de ferramenta gráfica no auxílio do uso do sistema de padrões de especificação de propriedades.

5.2 Conclusões e trabalhos futuros

Muito há que se fazer tanto em pesquisa quanto em implementação para que a tarefa de modelagem de sistemas em Redes de Petri Coloridas seja efetivamente bem auxiliada por técnicas e métodos de reúso de modelos. Neste trabalho, necessidades foram levantadas e algumas soluções foram detalhadas e implementadas servindo como um valioso ponto de partida.

Na etapa de construção e povoamento de repositórios de modelos, já se tem uma solução para construção do repositório usando os próprios modelos CPN. O método introduzido é funcional e pode ser melhorado através da construção de uma biblioteca de funções SML'97. Promovendo então, a automatização do processo de inserção de um novo modelo no repositório. Também se torna necessário que cada projetista envolvido em um projeto de modelagem de grande porte, usando o Design/CPN como ferramenta, identifique e armazene os modelos que implementem soluções que possam ser generalizadas e reusadas em diferentes contextos futuros. Quanto mais automatizado se tornar o método de armazenamento, mais facilmente ter-se-á uma boa base de modelos reusáveis.

Outro aspecto relevante é a integração dos modelos recuperados. Na implementação apresentada neste trabalho, esta tarefa manual é deixada totalmente a cargo do projetista. Da mesma forma, a adaptação de um modelo candidato a reúso recuperado, não foi abordada. Soluções para integração e adaptação são relevantes como sugestões para trabalhos futuros.

Quanto ao problema da construção de fórmulas em lógica temporal, apesar de nem sempre ser uma atividade trivial, há soluções para minorar esta sobrecarga cognitiva imposta ao projetista. O reúso de experiências de especificação de propriedades através de uso de um sistema de padrões de propriedades [DAC98] é uma delas. É fundamental entretanto, que se exercite mais o uso do ASK-CTL para que se possa escrever mapeamentos dos padrões existentes para o mundo de propriedades sob espaço de estados de Redes de Petri.

A recuperação de modelos pode ser melhorada através da integração do Design/CPN com ferramentas gráficas, assistentes de recuperação mais amigáveis que acelerem o mecanismo de busca reduzindo ainda mais o ônus de modelar sistemas complexos.

A implementação da busca automática de modelos dentro do repositório pode ser melhorada através de buscas paralelas, ou ainda através do uso de técnicas de redução do espaço de estados, como o uso de relações de equivalência [Jen97]. Tais soluções podem promover a diminuição do tempo de busca devido a paralelização ou pela geração de espaços de estados mais compactos para o repositório.

Quanto à integração do modelo recuperado com o modelo em construção, pode-se aumentar o nível de automatização na medida que se fornece mais riqueza de detalhes sobre o modelo recuperado. Ou seja, quanto mais detalhes sobre a estrutura do modelo recuperado, mais fácil torna-se programar a sua integração em um projeto em desenvolvimento.

Embora a adaptação de modelos recuperados não seja o foco de pesquisa deste trabalho, é possível faze-lo automaticamente através de um processo de síntese de um novo modelo a partir de restrições de comportamento. Tal trabalho está sendo realizado por Kyller Gorgônio como sua pesquisa de mestrado também no Laboratório de Redes de Petri e utiliza o algoritmo de síntese de controlados da teoria de controle supervisório e especificações de propriedades em Lógica Temporal para adaptar um modelo CPN [APKCG00].

Considerando a atividade de construção de modelos de sistemas de software como uma atividade de resolução de problemas, uma possível solução para gerenciar a complexidade da modelagem de sistemas complexos seria adotar uma abordagem multi-agentes. Neste sentido, o procedimento para recuperação de modelos candidatos a reúso poderia ser utilizado para implementar casamento de padrões com o objetivo de identificar agentes em uma sociedade de agentes incluindo habilidades específicas para solução de determinados problemas de modelagem em domínios específicos. De fato, alguns trabalhos já têm sido desenvolvidos com esta intenção, como por exemplo no contexto de ensino de redes de Petri [Gói00; CGdFP98] e projeto de sistemas de transporte [CPdF96].

Em suma, este trabalho contribuiu não só com uma pesquisa a respeito dos mecanismos de reúso aplicados à etapa de modelagem de sistemas de software (mais especificamente usando Redes de Petri Coloridas), mas também com algumas propostas de solução aos problemas levantados dentro do tema geral e com algumas soluções particulares a problemas de ordem prática.

Bibliografia

[AB96] W.M.P. van der Aalst and T. Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. Computing Science Reports 96/06, Eindhoven University of Technology, Eindhoven, 1996.

[APKCG00] Angelo Perkusich and Kyller Costa Gprgônio. Síntese de Especificações em Redes de Petri para Componentes de Software. In *XIV SBES - Workshops*, pages 139 – 144, João Pessoa, PB - Brasil, October 2000.

[BP99] T.C. Barros and A. Perkusich. Design of supervisors for agile manufacturing systems using colored petri nets. In *15th International Conference on CAD/CAM Robotics and Factories of the Future, CARS and FOF'99*, volume 1, pages MF1–MF6, Águas de Lindóia, SP, Brasil, August 1999.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[CCM96] Allan Cheng, Søren Christensen, and Kjeld H. Mortensen. Model checking coloured petri nets exploiting strongly connected components. In *In M.P. Spathopoulos, R. Smedinga, and P. Kozák, editors. Proceedings of the International Workshop on Discrete Event Systems, WODES96. Institution of Electrical Engineers, computing and control division, 19-21 August 1996. Edinburgh, Scotland, UK.*, pages 169–177, 1996.

[CGdFP98] E.B. Costa, G.M. Góis, J.C.A. de Figueiredo, and A. Perkusich. Towards a multi-agent interactive learning environment oriented to the petri net domain. In *Proc. of IEEE Int. Conf. on Systems Man and Cybernetics*, pages 250–261, San Diego, USA, October 1998.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM - TOPLAS*, 16(5), September 1994.

[CGL96] E. Clarke, O. Grumberg, and D. Long. Model checking. *Springer-Verlag Nato ASI Series F*, 152, 1996. a survey on model checking, abstraction and composition.

[CJK97] S. Christensen, J. B. Joergensen, and L. M. Kristensen. Design/CPN — A computer tool for coloured Petri nets. *Lecture Notes in Computer Science*, 1217:209–221, 1997.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, 1973.

[Cle95] Paul Clements. From subroutines to subsystems: Component-based software development. *American Programmer*, 8, November 1995.

[CPdF96] E.B. Costa, A. Perkusich, and J.C.A. de Figueiredo. A multi-agent environment to aid in the design of petri nets based software systems. In *Proc. of The Eighth International Conference on Software Engineering and Knowledge Engineering, SEKE'96*, pages 253–259, Lake Tahoe, USA, June 1996.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New YOrk, March 1998. ACM Press.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press, ACM Press, 1999.

[Dus92] E. Dusink. Reuse is not done in a vacuum. In *WISR'92, 5th Annual Workshop on Software Reuse*, Palo Alto, California, October 1992.

[DV94] B. Blongard D.ribot and C Villermain. Development life-cycle with reuse. In *ACM Symposium on Applied Computing SAC 94*, March 1994.

[Fre83] P. Freeman. Reusable software engineering: Concepts and research directions. In *Workshop on Reusability in Programming*, pages 2–16, September 1983.

[FS97] Bernd Fischer and Gregor Snelting. Reuse by contract. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26 1997*, pages 91–100, September 1997.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.

[Gar94] David Garlan. Integrating formal methods into a professional master of software engineering program. In *Proceedings of the Eighth Z User Meeting*, Workshops in Computing, Cambridge, England, June 1994. Springer-Verlag.

[GD90] D Garlan and N. Delisle. Formal specification as reusable frameworks. In *VDM 90: VDM and Z — Formal Methods in Software Development. 3rd International Symposium of VDM Europe. LNCS 428*, pages 150–163, 1990.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[Gói00] G. M. Góis. Um sistema tutor multi-agentes no domínio de redes de petri. Dissertação de mestrado, Curso de Mestrado em Informática - Universidade Federal da Paraíba, Campina Grande, Paraíba, 2000.

[GJM91] Carlo Gezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International Editions, Englewood Cliffs, New Jersey 07632, 1991.

[Gra95] Bernd Grahlmann. PEP: A Programming Environment based on Petri Nets. Tool Presentation of ATPN'95, June 1995.

[Gra97] Bernd Grahlmann. The PEP Tool. In *Tool Presentations of ATPN'97 (Application and Theory of Petri Nets)*, June 1997.

[Gue97] Dalton Dario Serey Guerrero. Sistemas de redes de petri modulares baseadas em objetos. Dissertacao de mestrado, COPIN - Universidade Federal da Paraiba, 1997.

[Hem96] Thomas Hemmann. Thomas hemmann: On the reuse of software engineering reuse approaches and techniques in knowledge engineering, 1996.

[HM84] E. Horowitz and J. Munson. An expansive view of reusable software. *IEEE Transactions on Software Engineering*, 10(5):477–487, September 1984.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use*. EACTS – Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[Jen97] K. Jensen. *Coloured petri nets-Basic Concepts, Analysis Methods and Practical Use*, volume 2. Springer-Verlag, 1997.

[JRBM92] E. Clarke J. R. Burch and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98, June 1992.

[Knu68] D. E. Knuth. *The Art of Computer Programming I: Fundemental Algorithms*. Addison–Wesley, Reading, Massachusetts, 1968.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[Lak95] C. Lakos. From coloured petri nets to object petri nets. In *Application and Theory of Petri Nets*, volume 935, pages 278–297, Torino, Italy, June 1995.

[LHCB98] N.-H. Lee, J.-E. Hong, S.-D. Cha, and D.-H. Bae. Towards reusable colored petri nets. In *Proc. Int. Symp. on Software Engineering for Parallel and Distributed Systems, 20-21 April 1998, Kyoto, Japan*, pages 223–229, 1998.

[MA81] Clarke E. M. and Emerson E. A. Characterizing properties of parallel programs as fixpoints. In *Seventh International Colloqium on Automata, Languages and Programming*, volume 85, 1981.

[Mac92] K.L. MacMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 1992.

[McI69] M. D. McIlroy. "Mass produced" software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1969. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

[MMM95] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989.

[NE68] P. Nauer and B. Randell Eds. Software engineering; report on a conference by the NATO science committee. Technical report, NATO Scientific Affairs Division, Brussels, Belgium, 1968.

[Nei94] J. M. Neighbors. Reuse so far: Phasing in a revolution. In *Third International Conference on Software Reuse*, pages 191–192, November 1994.

[NGT92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, September 1992.

[NJ99] Martin Naedele and Jorn W. Janneck. Design patterns in petri net system modeling. October 1999.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science*, pages 46–57, 1977.

[SCP98] J.C.A. de Figueiredo S.A.D. Costa, D.D.S. Guererro and A. Perkusich. Inheritance issues in object-oriented petri net modeling. In *IEEE Int. Conf. on Systems Man and Cybernetics*, pages 196–201, San Diego, USA, October 1998.

[Sti89] C. Stirling. Comparing linear and branching time temporal logics. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of the Conference on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 1–20, Berlin, April 1989. Springer.

[Tra95] Will Tracz. Confessions of a used-program salesman: Lessons learned. In *Symposium on Software Reusability*, pages 11–13, April 1995.

[Ull93] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1993.

[Uni96] University of Aarhus. *Design/CPN ASK-CTL Manual*, 1996.

[Weg89] P. Wegner. Capital-intensive software technology. pages 43–97, 1989.

[WVF97] Jeannette M. Wing and Mandana Vaziri-Farahan. A case study in model checking software systems. *Science of Computer Programming*, 28, 1997.

[ZW93] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: A Key to Reuse. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 182–190, December 1993.

[ZW95a] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching, a tool for using software libraries. ACM Transactions on Software Engineering and Methodology (TOSEM), April 1995.

[ZW95b] Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 6–17, October 1995.