Adaptação de Modelos em Redes de Petri Coloridas

Kyller Costa Gorgônio

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

Resumo

Neste trabalho introduz-se uma abordagem para a adaptação automática de modelos em redes de Petri. Para esta abordagem define-se um procedimento que, a partir de um modelo e de um conjunto de restrições de comportamento, sintetiza um novo modelo. Este procedimento foi definido com base nos conceitos e técnicas da Teoria do Controle Supervisório e da Verificação Automática de Modelos. Além disso, aborda-se o problema da adaptação num contexto de reúso de modelos.

Abstract

In this work we introduce an automatic adaptation approach for Colored Petri Nets models. For this approach we define a procedure for the synthesis of a new model based on a given model and a set of behavior restrictions. This procedure was defined based on the concepts and techniques of the supervisory control theory and model checking. Moreover, we tackle the adaptation problem in the context of models reuse.

Agradecimentos

Agradeço a todos que, seja através de palavras ou de atitudes, mostraram-me quais os caminhos a serem seguidos e quais a serem evitados. Agradeço a todos os meus amigos e familiares que acompanharam-me durante esta jornada, e proporcionaram-me ótimos momentos.

Em especial agradeço a meus pais (Lúcia e Kyval), professores de profissão, que sabendo do valor da educação não mediram esforços para que eu pudesse concluir esta etapa da minha vida. Por fim, mas não menos importante, agradeço a minha avó materna, D. Guia do Educandário Plínio Lemos, que a mais de cinquenta anos tem se dedicado de corpo e alma à educação e a sua familia.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente, sistemas de hardware e software tem sido amplamente utilizados em situações em que falhas são inaceitáveis, tais como: comércio eletrônico, sistemas de telefonia, de controle de tráfego terrestre e aéreo, instrumentação médica, entre outros. Freqüentemente relatam-se incidentes em que falhas foram ocasionadas por erros nos sistemas de hardware ou de software. Um exemplo recente de tal falha foi a explosão, no dia 4 de junho de 1996, do foguete Ariane 5 menos de quarenta segundos após seu lançamento. A comissão que investigou as causas do acidente determinou que o mesmo foi ocasionado por um erro no software para os cálculos da trajetória do foguete. Em face disto, esta mesma comissão sugeriu que todo o software utilizado no Ariane 5 fosse verificado para prevenir outros acidentes similares.

Desta forma, é bastante claro que é necessário dispor de mecanismos que possibilitem o projeto de sistemas de software maiores, mais complexos e com maior grau de segurança em seu funcionamento [CW96]. Uma solução para gerenciar a complexidade associada à construção de artefatos de software é construir-los de maneira que possam ser reaproveitados em projetos futuros. Desta maneira, o processo de desenvolvimento de um novo sistema de software não mais começaria do zero, pois já existiriam alguns artefatos que poderiam ser reaproveitados.

Já em 1968 estes problemas foram detectados, e neste mesmo ano realizou-se

uma conferência patrocinada pela OTAN[1] em que foi proposta uma solução para o problema da *crise de software* baseada nos princípios de disciplinas de engenharia bem estabelecidas [Kru92]. Foi nesta conferência que surgiu a engenharia de software.

A idéia básica desta solução consistia em construir artefatos de software de maneira que estes pudessem ser reaproveitados em situações distintas daquelas para as quais foram inicialmente concebidos [McI69].

Mais de três décadas após esta conferência, as idéias de reúso de artefatos de software continuam aceitas como uma abordagem poderosa para promover práticas de engenharia de software [Kru92], diminuindo o esforço necessário para conceber novos sistemas de software.

De maneira simplificada, o reúso de software pode ser definido como o reúso de artefatos de software durante a construção de um novo sistema de software. Os tipos de artefatos que podem ser reusados não estão limitados apenas a trechos de código fonte, mas podem incluir decisões de projeto, especificações, documentação, etc [BP89b; BP89a; Fre87; Tra88].

Genericamente, os modelos de produção de sistemas de software são divididos em fases, a saber: análise, projeto, codificação, teste e manutenção. Entretanto, na maioria dos casos assume-se que o desenvolvimento de um novo sistema de software inicia sempre do zero. Assim, é necessário adequá-los para que as atividades de reúso sejam incorporadas ao processo de desenvolvimento.

Conceitos de reúso utilizados na fase de codificação podem ser introduzidos na fase de projeto [RBV94], e os objetos de reúso passam a ser modelos de sistemas em vez de código fonte. Se esses modelos são descritos em alguma linguagem formal, então é possível investigar o comportamento desses modelos de forma automática, incluindo análise, verificação e simulação [CW96].

Krueger [Kru92] constatou que as principais atividades num contexto de reúso são: *abstração*, *seleção*, *adaptação* (também chamada de especialização) e *integra-*

[1] Organização do Tratado do Atlântico Norte.

ção. No contexto deste trabalho o principal objetivo é definir um mecanismo automático para adaptação de modelos com base em um conjunto de restrições de comportamento.

O mecanismo de adaptação de modelos introduzido neste trabalho é baseado nos resultados da Teoria do Controle Supervisório (TCS) [RW89]. Neste contexto o objetivo é assegurar que o modelo obtido satisfaz o comportamento desejado com o mínimo de restrições possível. Na TCS os modelos são descritos através de geradores (autômatos) finitos [HH79], e admite-se que os eventos, alfabeto da linguagem reconhecida, que podem ser processados pelo sistema são divididos em controláveis e não-controláveis. Desta forma, o problema principal abordado no contexto da TCS é encontrar uma linguagem, minimamente restritiva, para a qual os estados do modelo considerados indesejados não são alcançados.

Um dos principais resultados da TCS é a definição do algoritmo da suprema sublinguagem controlável. Este algoritmo determina uma nova especificação a partir da especificação de um sistema e um conjunto de restrições de comportamento. De fato, esta nova especificação, no contexto deste trabalho, é adotada como o modelo adaptado, ou seja, o modelo a ser reusado.

Neste trabalho, trata-se do problema da modelagem de sistemas distribuídos e concorrentes. Devido à natureza desses sistemas, optou-se pela utilização das redes de Petri [Mur89] como ferramenta de modelagem formal.

As redes de Petri são um modelo matemático com uma notação gráfica associada, Adequadas à modelagem de sistemas assíncronos, distribuídos e concorrentes [Mur89]. A característica matemática possibilita simular, analisar e verificar modelos automaticamente. Além disso, sua notação gráfica promove uma melhora compreensão do projetista, com relação ao modelo do sistema, durante o processo de modelagem. Devido a complexidade dos sistemas tratados, serão utilizadas as redes de Petri Coloridas [Jen92] no desenvolvimento deste trabalho, pois estas possibilitam descrições mais compactas dos sistemas tratados.

Para adaptar modelos é preciso descrever um conjunto de propriedades que são

desejadas do modelo, as restrições de comportamento. Apesar das redes de Petri serem um formalismo adequado para a modelagem de sistemas concorrentes, não o são para a descrição de propriedades, o que pode ser feito de maneira mais natural utilizando linguagens orientadas a declarações, como linguagens algébricas e lógica temporal. No contexto deste trabalho optou-se por utilizar a lógica temporal ASK-CTL para especificar as propriedades, pois esta foi definida no contexto de redes de Petri Coloridas [CCM97].

Associada à utilização da lógica temporal estão os mecanismos de verificação automática de modelos. Estes mecanismos atuam sobre o espaço de estados do modelo determinando se uma determinada fórmula escrita em lógica temporal é, ou não, satisfeita pelo modelo. No caso das redes de Petri Coloridas o espaço de estados é o grafo de alcançabilidade que descreve o comportamento delas.

Tendo em mente o objetivo de implementar uma ferramenta que possibilite a adaptação automática de modelos em redes de Petri Coloridas, é preciso dispor de ferramentas que possibilitem editar, analisar e verificar estes modelos. Neste trabalho optou-se pela utilização do Design/CPN [JCHH91]. O Design/CPN é um conjunto de ferramentas bastante poderoso que permite manipular modelos em redes de Petri Coloridas. Entre as ferramentas que o compõem incluem-se: um editor gráfico, um simulador interativo, um gerador de grafo de ocorrência (espaço de estados), e uma ferramenta para análise de desempenho. Além dessas ferramentas há um conjunto de bibliotecas, dentre as quais destaca-se a biblioteca ASK-CTL [CM96] como a de mais de interesse no contexto deste trabalho. A biblioteca ASK-CTL disponibiliza mecanismos para escrever fórmulas e realizar a verificação automática destas no espaço de estados gerado pelo Design/CPN.

1.1 Objetivos

O principal objetivo deste trabalho é definir um mecanismo automático que permita ao projetista adaptar modelos de maneira que estes satisfaçam a um conjunto de

restrições de comportamento definidas.

Os modelos fornecidos para o mecanismo de adaptação são descritos em redes de Petri Coloridas, e as propriedades desejadas desse modelo são especificadas em lógica temporal. Desta forma é possível averiguar, utilizando um verificador automático de modelos, se o modelo fornecido satisfaz às propriedades de forma computacionalmente eficiente.

O mecanismo de adaptação de modelos introduzido é baseado nos resultados da Teoria do Controle Supervisório. Desta forma, busca-se assegurar que o modelo obtido irá satisfazer o comportamento desejado com o mínimo de restrições possível.

1.2 Escopo e Relevância

Neste trabalho estuda-se o reúso no contexto da modelagem de sistemas de software complexos, concorrentes e distribuídos, e sua importância reside na introdução destes conceitos no projeto de modelos formais, contribuindo para um ganho de produtividade na modelagem formal de sistemas.

Devido à natureza dos sistemas em questão, e à existência de boas ferramentas de modelagem com suporte às atividades de verificação e análise, optou-se por desenvolver este trabalho baseado em redes de Petri Coloridas.

A técnica de adaptação de modelos introduzida baseia-se nos conceitos da Teoria do Controle Supervisório, e sua aplicação resulta sempre em um modelo que satisfaz ao conjunto de restrições comportamentais. Caso as restrições não possam ser satisfeitas, o modelo obtido é um modelo vazio.

Entretanto, é importante observar que o algoritmo da suprema sub-linguagem controlável opera sobre o espaço de estados do modelo do sistema, e a explosão do tamanho do espaço de estados dos modelos é o maior problema encontrado quando da aplicação deste algoritmo. Assim, é necessário investigar mecanismos que permitam representar e manipular eficientemente este espaço de estados.

Através da utilização da técnica de verificação automática de modelos é possível

determinar se modelo fornecido satisfaz, ou não, as restrições de comportamento desejadas. Desta forma, evita-se a execução do algoritmo de síntese quando não é possível modificar o modelo para satisfazer as restrições.

1.3 Organização do Trabalho

Esta dissertação está organizada em cinco capítulos além deste: No Capítulo 2 são apresentados os conceitos básicos que suportam a abordagem de adaptação de modelos introduzida. No Capítulo 3 será discutido o reúso de modelos formais. No Capítulo 4 será apresentado a técnica de adaptação de modelos que é o foco principal deste trabalho. No Capítulo 5 será ilustrada a aplicação da técnica de adaptação introduzida em um contexto de reúso de modelos formais. E, por fim, no Capítulo 6 serão apresentadas as conclusões deste trabalho.

Capítulo 2

Fundamentação Teórica

Neste capítulo serão introduzidos os conceitos básicos dos formalismos a serem utilizados neste trabalho. Na Seção 2.1 serão apresentadas as redes de Petri, introduzindo os conceitos de redes de Petri Coloridas e de redes Hierárquicas. Na Seção 2.2 é apresentada a técnica de Verificação Automática de Modelos. E na Seção 2.3 apresenta-se a Teoria do Controle Supervisório, introduzindo as noções de linguagens controláveis e de supervisores.

2.1 Redes de Petri

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]. Possuem uma representação gráfica associada à sua notação que facilitam a interação entre os projetistas envolvidos na construção de um sistema de software.

Uma rede de Petri é composta por 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, de acordo com uma *regra de ocorrência*, estabelece a sua semântica [dM00].

Como observa-se na Figura 2.1, a estrutura de uma rede de Petri é representada

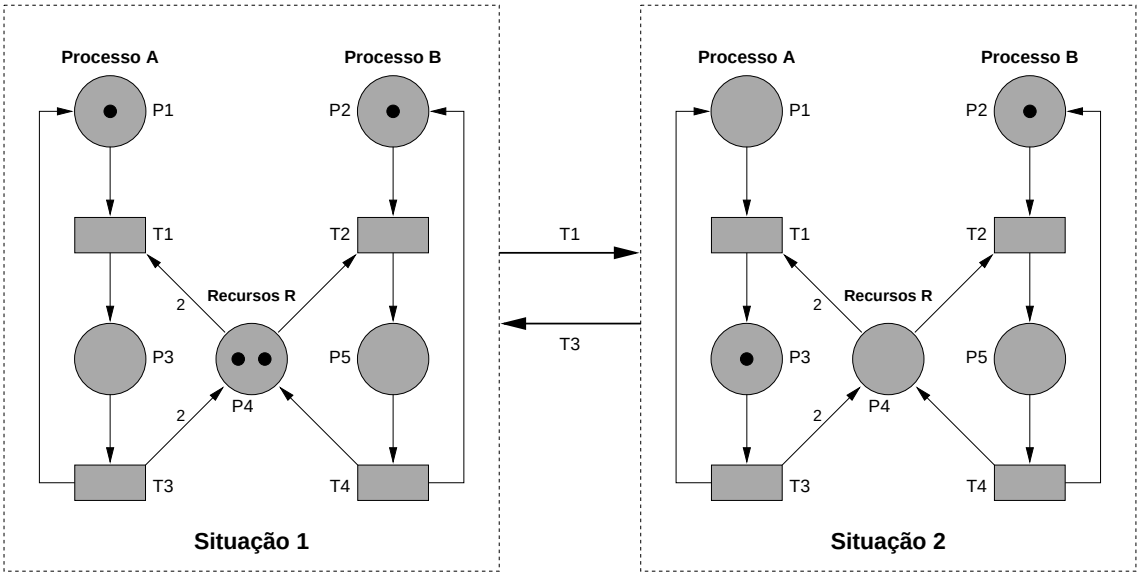

Figura 2.1: Exemplo de rede de Petri Lugar/Transição.

por um grafo bipartido direcionado, cujos nós são de dois tipos: *lugares* ou *transições*. Os lugares podem representar estados ou recursos e as transições ações ou eventos.

Formalmente, 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 são representados por círculos, as transições por retângulos e a relação de fluxo por arcos direcionados.

Na rede de Petri apresentada na Figura 2.1 as inscrições definem o *peso dos arcos*. Por exemplo, ao arco que liga **T3** a **P4** está associado o peso **2**.

A marcação de uma rede de Petri determina o seu estado. Uma marcação corresponde à associação de elementos, denominados *fichas*, aos lugares. Por exemplo, na Figura 2.1 as fichas existentes nos lugares da **Situação 1**, estabelecem a sua marcação (estado) inicial. Como o estado dos sistemas representados pelas redes de Petri normalmente[1] varia à medida que eventos ocorrem, existe uma regra de ocorrência para determinar a evolução das marcações. Essa regra estabelece as condições para que uma transição esteja *habilitada a ocorrer* e quais as conseqüências de sua ocorrência.

Por exemplo, a rede de Petri Lugar/Transição apresentada na Figura 2.1 modela uma situação na qual dois tipos de processos, **A** e **B**, utilizam recursos do tipo **R**. O processo **A** precisa de dois recursos **R** para executar, enquanto o processo **B** necessita apenas de um recurso **R**.

A semântica de uma rede de Petri é definida por sua regra de disparo. Para uma rede de Petri Lugar/Transição tem-se que:

1. uma transição está habilitada se cada um de seus lugares de entrada contém um número de fichas maior ou igual ao especificado no peso do respectivo arco de entrada;

2. uma transição habilitada pode ocorrer ou não;

3. quando uma transição habilitada ocorre são removidas de seus lugares de entrada a quantidade de fichas indicada em seus respectivos arcos de entrada e são adicionadas fichas nos lugares de saída observando o peso dos respectivos arcos de saída.

Ainda no exemplo da Figura 2.1, a transição **T1** tem como lugares de entrada **P1** e **P4** e, como lugar de saída **P3**. Na **Situação 1** duas transições estão habilitadas: **T1** e **T2**. Quando do disparo de **T1**, são removidas duas fichas de **P4** e uma ficha de **P1** e é colocada uma ficha em **P3**, sendo atingida a **Situação 2**.

[1] Teoricamente é possível que um evento ocorra e o estado do sistema permaneça inalterado.

2.1.1 Classes de Redes de Petri

Atualmente, o termo redes de Petri é usado genericamente para referenciar um conjunto de modelos que podem ser agrupados em classes.

Os primeiros modelos de redes de Petri pertencem à classe das redes de Petri de Baixo Nível, nas quais as inscrições associadas aos arcos da estrutura da rede definem apenas os pesos desses arcos, as fichas são indistintas e a ocorrência das transições é instantânea. Nessa classe, destacam-se as redes de Petri Elementares [Thi87] e as redes de Petri Lugar/Transição [Mur89].

Entretanto, quando usadas para modelar sistemas complexos (ou grandes), as redes de Baixo Nível apresentam algumas restrições. Uma delas é a necessidade de duplicação da estrutura de rede para modelar processos semelhantes ou idênticos. Como exemplo, observe que na rede Lugar/Transição da Figura 2.1 a sub-rede para o processo **A** tem estrutura igual àquela do processo **B**. Essa replicação decorre do fato de ser impossível diferenciar os processos por meio de fichas.

Uma outra limitação das redes de Baixo Nível é a carência de elementos para estudo de desempenho dos sistemas modelados, pois a ocorrência de suas transições é instantânea.

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 [MBC+95] 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 informação complexa 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 de redes de Petri de Alto Nível, as mais notáveis são as redes Predi-

cado/Transição [Gen87] e as redes de Petri Coloridas [Jen92]. No contexto deste trabalho os modelos sobre os quais investigam-se mecanismos de reúso são descritos em redes de Petri Coloridas. Assim, a seguir serão apresentados os conceitos básico destas redes.

2.1.2 Redes de Petri Coloridas

As redes de Petri Coloridas pertencem à classe das redes de Petri de Alto Nível, sendo muito utilizadas na modelagem de aplicações complexas. Uma definição formal detalhada para as redes de Petri Coloridas pode ser encontrada em [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, de maneira similar à definida para as redes de Petri Lugar/Transição. As declarações definem conjuntos de cores (domínios), variáveis e operações (funções) usadas nas inscrições. As inscrições, por sua vez, podem ser de quatro tipos: cores dos lugares, guardas, expressões dos arcos e inicializações [dM00].

As cores dos lugares determinam a cor (domínio) associada ao lugar. Um lugar só pode comportar fichas cujos valores respeitem sua cor. As guardas são expressões booleanas que restringem a ocorrência das transições. As expressões dos arcos servem para manipular a informação contida nas fichas. E as Inicializações são associadas aos lugares para estabelecer a marcação inicial da rede.

Um exemplo de uma rede de Petri Colorida é apresentado na Figura 2.2. As declarações estão expressas na caixa de linhas tracejadas no canto superior esquerdo. Os textos, em itálico, próximos aos lugares indicam suas cores e as expressões em negrito, suas inicializações. As expressões dos arcos localizam-se junto aos arcos direcionados e inexistem guardas associadas a transições.

Para compreender a regra de ocorrência para redes de Petri Coloridas, é essencial, primeiramente, compreender os conceitos de *variável de transição*, *ligação*[2] e *elemento de ligação*. As variáveis de transição são aquelas presentes nos arcos direci-

[2]Tradução para *binding*.

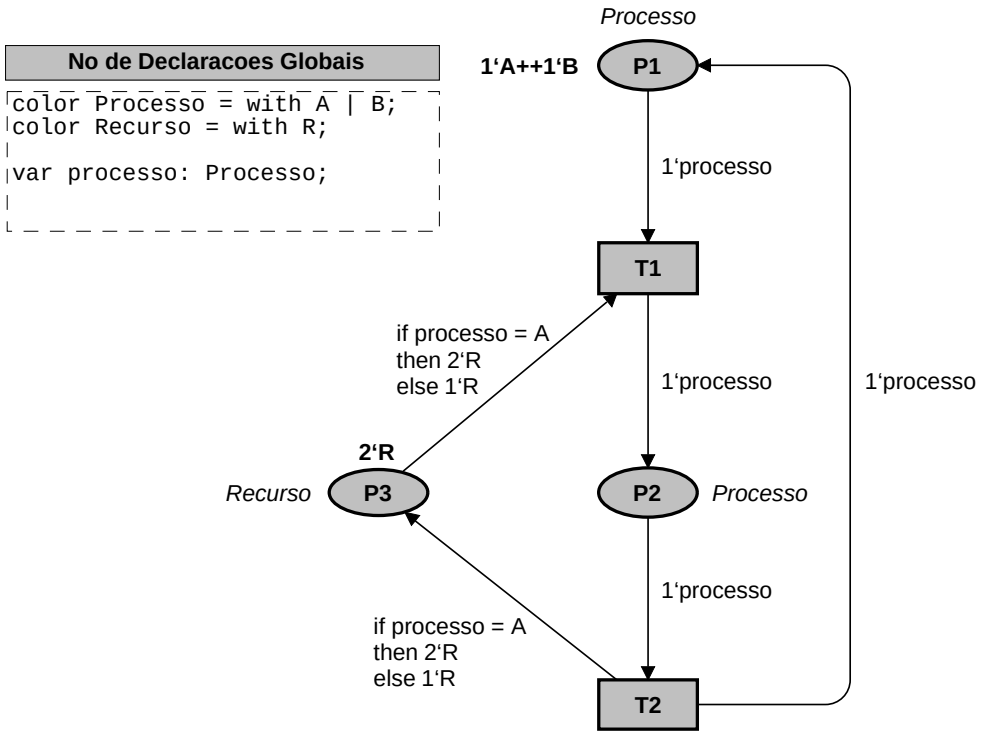

Figura 2.2: Exemplo de rede de Petri Colorida.

onados. Por exemplo, na Figura 2.2, a transição **T1** tem **processo** como sua variável de transição. Uma ligação corresponde a associação de uma variável de transição a um valor da sua cor. Por exemplo, para a mesma variável de transição **processo**, são possíveis as ligações $b_1 = \langle \textbf{processo} = \textbf{A} \rangle$ e $b_2 = \langle \textbf{processo} = \textbf{B} \rangle$. Um elemento de ligação é um par (transição, ligação). Como ilustração, considerando os exemplos anteriores, temos $be_1 = (\textbf{T1}, b_1 = \langle \textbf{processo} = \textbf{A} \rangle)$ e $be_2 = (\textbf{T1}, b_2 = \langle \textbf{processo} = \textbf{B} \rangle)$ como elementos de ligação.

A rede de Petri Colorida da Figura 2.2 modela uma situação idêntica àquela exemplificada na Figura 2.1. Entretanto, uma mesma estrutura foi usada para modelar os dois processos **A** e **B**, pois eles podem ser diferenciados através das fichas. As inscrições de inicialização dessa rede estabelecem que sua marcação inicial contém um processo do tipo **A**, um processo do tipo **B** e dois recursos tipo **R**. Conseqüentemente, apenas a transição **T1** está habilitada a ocorrer, pois ela é a única a apresentar algum elemento de ligação satisfeito pela marcação inicial. Nessa

situação, pode ocorrer be_1 ou be_2. Se be_1 ocorrer, *uma* ficha **A** é removida de **P1**, *duas* fichas **R** de **P3** e *uma* ficha **A** é adicionada a **P2** (pois em be_1 **processo** está ligada a **A**). Vale ressaltar que um elemento de ligação só está habilitado se satisfizer a guarda da transição. Assim, se para a transição **T1** fosse associada a guarda [**processo** = **A**], apenas be_1 estaria habilitado a ocorrer.

2.1.3 Redes de Petri Coloridas Hierárquicas

A idéia básica das redes de Petri Coloridas Hierárquicas (CPNH) é possibilitar a construção de um modelo através da combinação de um conjunto de redes relativamente menores denominadas *páginas*, de forma análoga à construção de um programa a partir de um conjunto de módulos e funções [Jen92]. O poder de modelagem de uma rede de Petri, de uma CPN e de uma CPNH são equivalentes. É sempre possível traduzir uma CPNH para uma CPN não hierárquica, que por sua vez pode ser traduzida para uma rede de Petri. Em termos de linguagem de programação, podemos comparar as redes de Petri com as linguagens de máquina, as CPNs com a introdução de elementos de dados estruturados na programação e as CPNHs com o uso de módulos e funções.

As CPNHs são construídas utilizando-se o conceito 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 de um dos lugares, uma ficha idêntica é adicionada ou removida de todos os outros lugares pertences ao conjunto. Um conjunto de lugares de fusão é denominado conjunto de fusão (*fusion set*). Uma transição de substituição pode ser vista como uma transição de mais alto nível que se relaciona a uma rede mais complexa e que fornece maiores detalhes das atividades representadas pela transição de substituição. A página que contém a transição de substituição é denominada *superpágina* e a que contém uma visão mais detalhada é denominada *subpágina*. Cada transição de substituição é denominada *supernó* da subpágina correspondente. 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 forma um conjunto de fusão. Dessa forma, quando uma ficha é depositada num *socket*, ela aparece também na porta associada aquele *socket*, permitindo assim a conexão entre a superpágina e a subpágina. É sempre possível traduzir uma CPNH para sua correspondente não hierárquica. Para isso, basta substituir cada transição de substituição e arcos conectados, por sua respectiva subpágina "colando" cada *socket* com sua respectiva porta. A definição formal de CPNH pode ser encontrada em [Jen92].

2.2 Verificação Automática de Modelos

Os principais métodos de validação de sistemas complexos são simulação, teste, verificação dedutiva e verificação automática de modelos. Simulação e teste [Mye79] envolvem a realização de experimentos antes do software ser terminado. Enquanto a simulação é realizada sobre uma abstração, ou modelo, do sistema, testes são realizados sobre a implementação deste sistema. Apesar de serem bastante úteis para encontrar erros, raramente é possível verificar todas as possibilidades existentes através da utilização destas técnicas [CJGP99].

O termo verificação dedutiva refere-se ao uso de axiomas e métodos de prova para verificar a corretude dos sistemas. Há algum tempo atrás, assumia-se que a importância de verificar tal corretude era tão elevada que o desenvolvedor poderia utilizar o tempo que fosse preciso para fazê-lo. Inicialmente, tais provas eram realizadas manualmente. Depois foram construídas ferramentas de software para agilizar esta tarefa, tais como a verificação automática de modelos.

A verificação automática de modelos [CGL94; CGL96] é uma técnica utilizada na verificação de sistemas reativos com comportamento finito, tais como circuitos

seqüenciais e protocolos de comunicação [CGL96]. As propriedades do sistema a ser verificado são descritas em uma lógica temporal proposicional, e seu modelo é dado por um autômato. Um procedimento de busca eficiente é utilizado para determinar automaticamente se as propriedades são satisfeitas pelo modelo.

Essa técnica apresenta algumas vantagens relevantes sobre os mecanismos de prova de teoremas [CW96] para a verificação de circuitos e protocolos. A mais importante delas é que o procedimento é completamente automático. Genericamente, o usuário provê uma representação de alto nível do modelo e um conjunto de propriedades a serem verificadas. O verificador automático de modelos[3] termina sua execução gerando na saída uma resposta *verdade*, significando que o modelo satisfaz às propriedades, ou fornecendo um contra-exemplo em que a propriedade não é satisfeita. Os contra-exemplos são particularmente importantes na depuração de sistemas reativos complexos.

Os primeiros verificadores automáticos de modelos eram capazes de detectar erros em circuitos e protocolos [CES86]. Entretanto, devido ao problema da explosão do espaço de estados, era inviável verificar sistemas complexos. Devido a esta limitação, especulou-se que a verificação automática de modelos jamais poderia ser utilizada na prática.

A possibilidade de verificar sistemas complexos surgiu no início dos anos 80 com a definição de estruturas de dados que possibilitassem representar relações de transições utilizando *Diagramas Binários de Decisão Ordenados* (*Ordered Binary Decision Diagrams - OBDDs*) [Bry86]. A idéia dos OBDDs é relativamente simples. Basicamente, assume-se que o comportamento de um sistema é determinado por um conjunto de n variáveis booleanas: $v_1, v_2, \ldots, v_n$. Então a relação de transição do sistema pode ser expressa através de uma fórmula booleana:

$$R(v_1, v_2, \ldots, v_n, v'_1, v'_2, \ldots, v'_n)$$

onde $v_1, v_2, \ldots, v_n$ representam o estado atual do sistema, e $v'_1, v'_2, \ldots, v'_n$ representam o próximo estado. Convertendo esta fórmula em uma OBDD, é obtida uma

[3] Tradução para *model checker*.

representação compacta da relação de transição.

A aplicação do algoritmo original de verificação automática de modelos sobre esta nova representação para relações de transições, é chamada de *verificação de modelos simbólicos*[4]. Através dessa combinação tornou-se possível verificar sistemas extremamente complexos. De fato, alguns exemplos com mais de 10^{120} estados já foram verificados [BCL91; BCL$^+$94]. Isto é possível apenas porque o número de nós que precisam ser construídos no OBDD não mais depende do número de estados ou do tamanho do sistema de transições. Devido a este fato é possível verificar sistemas reativos realmente complexos.

2.2.1 Diagramas Binários de Decisão Ordenados

Os diagramas binários de decisão ordenados (OBDDs) são uma forma canônica para fórmulas booleanas introduzidos por Bryant [Bry86]. Eles são substancialmente mais compactos que os diagramas de decisão tradicionais, e podem ser eficientemente manipulados. Uma OBDD é semelhante a uma árvore de decisão binária, entretanto sua estrutura é um grafo direcionado acíclico, além disso existe uma ordenação rígida na seqüência de ocorrência das variáveis sempre que se atravessa o grafo da raiz até as folhas.

Considere, por exemplo, o OBDD apresentado na Figura 2.3, que representa a fórmula $(a \wedge b) \vee (c \wedge d)$, utilizando a ordenação de variáveis $a < b < c < d$. Associando valores às variáveis a, b, c e d é possível decidir se esta associação torna a avaliação da fórmula verdadeira (1) ou falsa (0). Por exemplo, para os valores $a = 1, b = 0, c = 1, d = 1$ chegasse a um nó cujo rótulo é **1** (verdade), logo a fórmula é avaliada como verdadeira. Bryant mostrou que dada uma ordenação das variáveis, existe um OBDD canônica para qualquer fórmula. O tamanho do OBDD depende desta ordenação nas variáveis.

[4]Tradução para *symbolic model checking.*

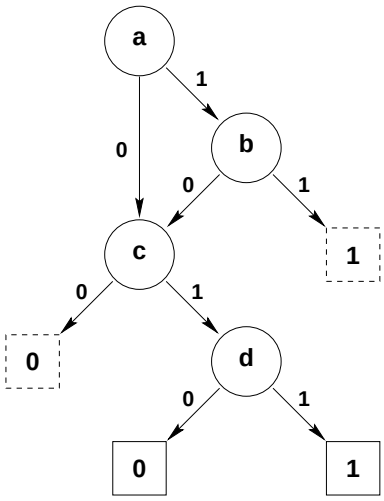

Figura 2.3: OBDD para a fórmula $(a \wedge b) \vee (c \wedge d)$.

2.2.2 Lógica Temporal

Em 1977, Amir Pnueli propôs a utilização da lógica temporal como base para provar a corretude dos sistemas concorrentes [Pnu77].

A lógica temporal, um tipo de lógica modal, é um sistema formal para descrever e verificar a ocorrência de eventos no tempo. Em qualquer sistema de lógica temporal são providos operadores para especificar como a veracidade de uma propriedade varia através do tempo. Tipicamente os operadores permitem expressar propriedades dos sistemas tais como: *invariância* (propriedades que são sempre verdadeiras), *eventualidade*[5] (a propriedade com certeza será verdadeira em algum instante futuro) e *precedência* (propriedade na qual um evento deve ocorrer antes de um outro). Assim, a lógica temporal provê um arcabouço útil para modelar sistemas de software, em especial aqueles com características não-determinísticas e concorrentes.

Existe um consenso entre pesquisadores e usuários que a lógica temporal constitui uma abordagem satisfatória para a descrição de propriedades de sistemas concorrentes [ES88]. Entretanto, este consenso não existe com relação a que tipo de lógica temporal para fazê-lo. Em sistemas de lógica temporal ramificada o tempo é visto

[5]No contexto deste trabalho, eventualmente significa que um evento certamente ocorrerá em algum instante.

como uma coleção parcialmente ordenada de instantes discretos e é permitido especificar diversos futuros alternativos. Em contraste, nos sistemas de lógica temporal linear o futuro é restrito a uma única possibilidade.

Lógica Temporal Ramificada

A lógica temporal ramificada[6] (CTL) [CE81; CES86] combina quantificadores de caminho com operadores da lógica linear. Quantificadores de caminho, **A** ("para todos os caminho") e **E** ("existe um caminho"), podem ser prefixos de uma combinação arbitrária dos operadores, **G** ("sempre") **F**, ("alguma vez"), **X** ("próximo"), **U** ("até que") e **V** ("a não ser que").

Existem dois tipos de fórmulas em CTL: *fórmulas de estado* (que são avaliadas para cada estado do sistema) e *fórmulas de caminho* (que são avaliadas através de um caminho). Seja AP o conjunto das proposições atômicas. A sintaxe das fórmulas de estado é dada pelas seguintes regras:

- Se $p \in AP$, então p é uma fórmula de estado.
- Se f e g são fórmulas de estado, então $\neg f$, $f \wedge g$ e $f \vee g$ são fórmulas de estado.
- Se f é uma fórmula de estado, então $E(f)$ é uma fórmula de estado.
- Se f é uma fórmula de estado, então $A(f)$ é uma fórmula de estado.

Adicionando mais duas regras descreve-se a sintaxe das fórmulas de caminho:

- Se f é uma fórmula de estado, então f também é uma fórmula de caminho.
- Se f e g são fórmulas de caminho, então $\neg f$, $f \wedge g$, $f \vee g$, Xf, fUg e fVg são fórmulas de caminho.

CTL é o conjunto de fórmulas geradas pelas regras acima descritas. Os quatro operadores mais utilizados em CTL são ilustrados na Figura 2.4. Cada árvore computacional tem o estado s_0 como raiz.

[6] Tradução para *Computation Tree Logics*.

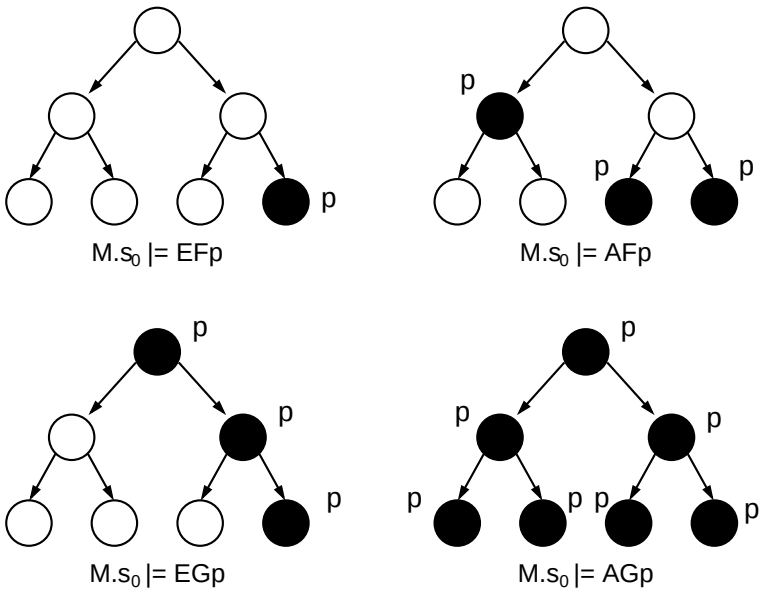

Figura 2.4: Operadores CTL básicos.

2.2.3 ASK-CTL

Como dito anteriormente, as redes de Petri Coloridas são apropriadas para a especificação de sistemas concorrentes, e suas propriedades são, geralmente, descritas em termos de seu espaço de estados [Jen92; Jen95], e lógicas temporais são boas para especificar propriedades de sistemas concorrentes [CES86]. ASK-CTL é uma lógica temporal, baseada em CTL, especialmente definida para expressar propriedades em termos do espaço de estados das redes de Petri Coloridas [CCM97; CM96]. No caso das redes de Petri Coloridas nas interpretações feitas sobre o espaço de estados, mais especificamente sobre sua representação por um grafo de ocorrência, consideram-se tanto informações dos nós como dos arcos deste grafo. Desta forma, em ASK-CTL são introduzidas duas categorias de fórmulas (predicados): fórmulas de estado e de transição.

Para que a lógica seja de uso prático, é preciso ser capaz de verificar fórmulas eficientemente. Como dito anteriormente, o problema da explosão do espaço de estados pode tornar a verificação impraticável. Algumas soluções para esse problema baseiam-se na diminuição da representação do espaço de estados [Jen95]. Outras soluções concentram-se em otimizar o caminhamento feito através do espa-

ço de estados. ASK-CTL baseia-se no segundo método. O algoritmo de verificação de ASK-CTL evita um caminhamento exaustivo no espaço de estados através da utilização do grafo de componentes fortemente conectados[7] (grafo SCC) [CJ95; Jen95].

Em um grafo SCC, como ilustrado na Figura 2.5, cada nó representa um subconjunto dos nós do espaço de estados de uma determinada rede de Petri, e cada um desses nós pode ser alcançado a partir de qualquer outro nó do subconjunto. Estes subconjuntos são disjuntos, e representam um particionamento do espaço de estados. Existe um arco entre dois nós do grafo SCC se existir um arco entre dois nós do espaço de estados e cada um desses nós pertencer a nós distintos do grafo SCC.

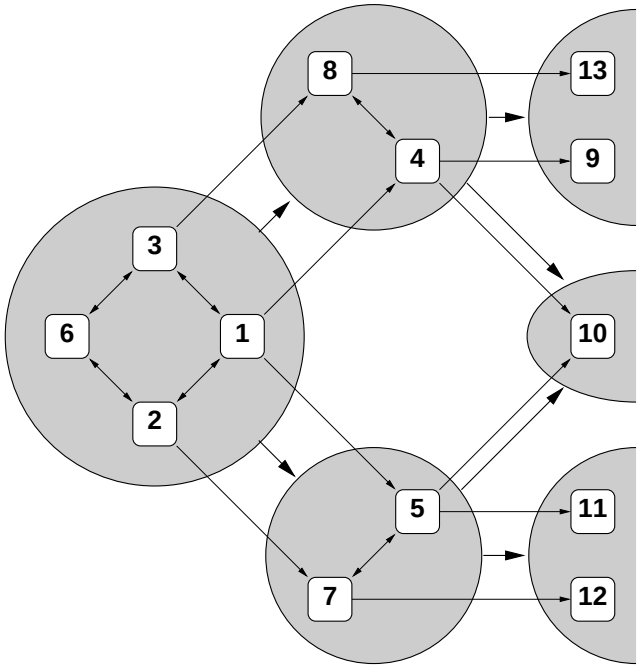

Figura 2.5: Exemplo de um grafo SCC.

[7] Tradução para *Strongly Connected Components*.

Sintaxe de ASK-CTL

Em ASK-CTL são definidas duas categorias mutuamente recursivas de fórmulas: *fórmulas de estado* e *de transição*.

Fórmulas de estado:

$$\mathcal{A} ::= tt \mid \alpha \mid \neg\mathcal{A} \mid \mathcal{A}_1 \vee \mathcal{A}_2 \mid \ < \mathcal{B} > \ \mid EU(\mathcal{A}_1, \mathcal{A}_2) \mid AU(\mathcal{A}_1, \mathcal{A}_2)$$

onde tt é interpretado como uma constante de valor "verdade", α é uma função que mapeia as marcações do espaço de estados em valores booleanos, e $\mathcal{B}$ é uma fórmula de transição. EU e AU serão descritos mais adiante.

Fórmulas de transição:

$$\mathcal{B} ::= tt \mid \beta \mid \neg\mathcal{B} \mid \mathcal{B}_1 \vee \mathcal{B}_2 \mid \ < \mathcal{A} > \ \mid EU(\mathcal{B}_1, \mathcal{B}_2) \mid AU(\mathcal{B}_1, \mathcal{B}_2)$$

onde β é uma função que mapeia os elemento de ligação em valores booleanos, e $\mathcal{A}$ é uma fórmula de estado.

É utilizada a convenção de sempre iniciar a avaliação de uma fórmula com $\mathcal{A}$, logo todas as escritas em fórmulas ASK-CTL são fórmulas de estado, e fórmulas de transição apenas podem aparecer como sub-fórmulas. Além disso, quando da realização da verificação de modelos, isto é feito com relação ao estado inicial.

A sintaxe de ASK-CTL é bastante parecida com a de CTL, exceto pelo operador $< \cdots >$. Este operador possibilita alternar entre fórmulas de estado e de transição.

Além dos operados booleanos $\neg$ e $\vee$, a lógica ASK-CTL contem o operador U ("até que") combinado com os quantificadores de caminho E e A ("existe" e "para todos" respectivamente). Por exemplo, o operador $EU(\mathcal{A}_1, \mathcal{A}_2)$ expressa a existência de um caminho no qual, a partir de uma dada marcação, a propriedade $\mathcal{A}_1$ é satisfeita até que seja alcançada uma em que $\mathcal{A}_2$ é satisfeita. Analogamente, $AU(\mathcal{A}_1, \mathcal{A}_2)$ requer que esta propriedade seja satisfeita em todos os caminhos.

Nenhuma restrição é feita com relação a computabilidade das funções booleanas α e β. Assume-se que esses predicados podem ser computados eficientemente para fins de verificar modelos.

A sintaxe de ASK-CTL é mínima, o que é vantajoso no momento de definir sua semântica formal. Entretanto, para aumentar a legibilidade de suas fórmulas são definidas abreviações para algumas fórmulas:

- $Pos\mathcal{A} \equiv EU(tt, \mathcal{A})$ É possível alcançar um estado em que $\mathcal{A}$ é verdade.
- $Inv\mathcal{A} \equiv \neg Pos\neg\mathcal{A}$ $\mathcal{A}$ é verdade em todos os estados alcançáveis, ou seja, $\mathcal{A}$ é invariante.
- $Ev\mathcal{A} \equiv AU(tt, \mathcal{A})$ Para todos os caminhos, $\mathcal{A}$ será verdade em um número finito de passos.
- $Along\mathcal{A} \equiv \neg Ev\neg\mathcal{A}$ Existe um caminho que ou é infinito ou termina em uma marcação morta, através do qual $\mathcal{A}$ é verdade em todos os estados.
- $< \beta > \mathcal{A} \equiv < \beta \wedge < \mathcal{A} >>$ Existe um estado sucessor M' em que $\mathcal{A}$ é verdade e $\mathcal{B}$ é verdade para o arco entre o estado atual e M'.
- $EX(\mathcal{A}) \equiv < tt\mathcal{A} >$ Existe um estado sucessor em que $\mathcal{A}$ é verdade.
- $AX(\mathcal{A}) \equiv \neg EX(\neg\mathcal{A})$ $\mathcal{A}$ é verdade para todos os estados sucessores, se existir algum.

De maneira semelhante, são definidas abreviações para as fórmulas de transição. O leitor interessado em mais detalhes pode consultar [CM96].

2.3 Teoria do Controle Supervisório

Nesta seção apresenta-se a Teoria de Controle Supervisório (TCS), desenvolvida em sua forma básica por Ramadge e Wonham [RW89; RW87a; RW87b] para a modelagem de sistemas a eventos discretos (SED).

Um SED pode ser representado por um gerador G tal que a linguagem gerada $L(G)$ seja L e a linguagem marcada $L_m(G)$ seja L_m. No entanto, sabe-se da teoria de linguagens [HH79] que nem todas as linguagens podem ser representadas por

autômatos ou geradores finitos. Além disso, é provado que a classe de linguagens representada por autômatos, ou geradores finitos é a classe de linguagens regulares.

Toda a teoria e resultados da TCS, desenvolvidos por Ramadge e Wonham, são baseados na teoria de linguagens e autômatos, portanto, os modelos de SEDs apresentados não se limitam àqueles representados por geradores finitos, embora os resultados computacionais mais precisos e definitivos se limitem àqueles representados por geradores finitos.

Tomando como exemplo a utilização de um recurso (por exemplo uma máquina) por um usuário, é obtido o modelo apresentado na Figura 2.6. Neste modelo o gerador G representa uma máquina com três estados possíveis: **I** (*inativo*), **U** (*em uso*) e **M** (*em manutenção*). As transições entre os estados são identificadas pelos eventos do alfabeto $\Sigma = \{\alpha, \beta, \lambda, \mu\}$. A máquina inicialmente está em repouso e quando é requisitada passa para o estado *em uso* através do evento α. Quando em uso, a máquina pode retornar ao seu estado inativo, após o término de sua tarefa, através do evento β, ou pode se danificar, passando ao estado *em manutenção* através do evento γ. Após a manutenção, a máquina retorna ao estado *inativo* através do evento μ.

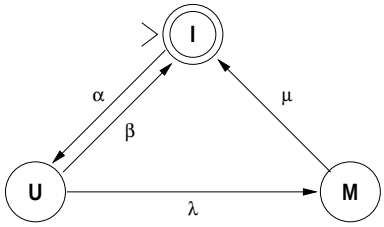

Figura 2.6: Utilização de um recurso por um usuário.

A linguagem gerada, $L(G)$, é o conjunto de todas as palavras geradas a partir do estado inicial **I**. Assim, a linguagem $L(G)$ é representada da seguinte forma:

$$L(G) = (\alpha\beta + \alpha\lambda\mu)^*(\epsilon + \alpha + \alpha\lambda).$$

Como o único estado marcado de G é o inicial, então a linguagem marcada $L_m(G)$ é formada por todas as palavras que representam um ciclo completo no grafo, ou

seja:

$$L_m(G) = (\alpha\beta + \alpha\lambda\mu)^*.$$

Note que $L(G)$ representa o comportamento fisicamente possível do sistema modelado, enquanto $L_m(G)$ representa as tarefas a serem realizadas pelo sistema.

2.3.1 Geradores Controlados

O modelo da Figura 2.6, é simplesmente um gerador espontâneo de cadeias de eventos, sem um controle externo. No entanto, muitas vezes é desejável que o sistema realize uma tarefa específica para a qual a ocorrência de algumas seqüências de eventos fisicamente possíveis, deve ser inibida. Para tanto, é necessário uma ação de controle externa sobre o sistema. Para modelar esta ação de controle, é necessário admitir que alguns eventos do sistema podem ser desabilitados quando desejado. Assim, particiona-se o alfabeto Σ em eventos *controláveis* Σ_c e eventos *não controláveis* Σ_u, de modo que:

$$\Sigma = \Sigma_c \cup \Sigma_u \text{ e } \Sigma_c \cap \Sigma_u = \varnothing$$

ou seja, os eventos em Σ_c podem ser desabilitados em qualquer instante de tempo, enquanto aqueles em Σ_u não sofrem influência da ação de controle. Essa partição do alfabeto de eventos reflete características de sistemas reais, como por exemplo a quebra de uma máquina em um sistema de manufatura é um evento não controlável, enquanto o início de operação de uma máquina é um evento controlável.

Assim como na teoria de controle clássica, denomina-se *planta* o modelo do sistema a ser controlado. A linguagem gerada é então denominada *linguagem da planta* e representa o comportamento do sistema na ausência de qualquer ação de controle. Uma ação de controle é dada através de uma entrada de controle. Denomina-se entrada de controle o conjunto de eventos habilitados em um determinado estado do sistema.

Quando se aplica uma entrada de controle γ a uma planta, esta se comporta como se os eventos inibidos fossem momentaneamente apagados de sua estrutura de

transição. Assim, a função de transição δ de um gerador, cujo alfabeto é particionado em eventos controláveis e não controláveis, não está definida para os eventos inibidos por uma dada entrada de controle aplicada ao gerador em um dado instante. Nesse sentido, o mecanismo de controle adotado pela TCS consiste exatamente em chavear as entradas de controle em resposta à seqüência de eventos gerada pelo sistema, de modo que a linguagem gerada sob a ação de controle especifique um conjunto de tarefas a serem realizadas.

Considerando o gerador da Figura 2.6, suponha que o início de cada ciclo de operação (evento α) e o término da manutenção (evento μ) sejam eventos controláveis, ou seja: $\Sigma_c = \{\alpha, \mu\}$ e $\Sigma_u = \{\beta, \lambda\}$. Nesse caso, o conjunto de entradas de controle válidas são $\Gamma = \{(\beta, \lambda), (\alpha, \beta, \lambda), (\beta, \lambda, \mu), (\alpha, \beta, \lambda, \mu)\}$.

Embora esse mecanismo de controle seja intuitivamente simples, na prática é necessário que se encontrem as condições para a existência do controlador que faça o chaveamento das entradas de controle da planta, de modo que uma dada especificação seja satisfeita. Nesse sentido, apresenta-se a seguir a noção de supervisor.

2.3.2 Supervisores

Como dito anteriormente, a ação de controle de um SED G consiste em chavear a entrada de controle através das entradas de controle $\gamma, \gamma', \gamma'', \ldots \in \Gamma$, em resposta à cadeia de eventos previamente gerada por G. Essa ação de controle é realizada por um agente externo denominado de supervisor.

A Figura 2.7 ilustra o modelo do sistema composto em malha fechada, do gerador controlado G_c supervisionado pelo supervisor $\mathbf{S}$.

A ação de controle modifica as linguagens associadas ao gerador, visto que algumas seqüências de eventos, antes possíveis de ocorrer, agora podem estar inibidas pela ação de controle.

Assim, pode-se dizer que a linguagem controlada é simplesmente a parte da linguagem marcada original que *sobrevive* sob a ação do controle, ou seja, $L_c(\mathbf{S}/G)$ representa as tarefas que podem ser completadas sob supervisão.

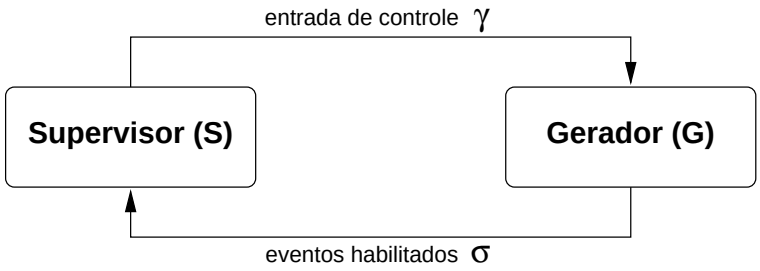

Figura 2.7: Supervisão de um SED.

Sob este aspecto, um dos principais resultados da TCS é a definição de um algoritmo que dados um gerador finito G, cujo alfabeto é composto por eventos controláveis e não-controláveis, e um conjunto de restrições de comportamento a serem aplicadas sobre G, encontre a suprema sub-linguagem controlável, denominada $supC(L)$, com relação à linguagem $L(G)$.

A linguagem $supC(L)$ representa todas as tarefas que, sob uma ação de controle, podem ser realizadas pelo sistema. Além disso, $supC(L)$ é a maior sub-linguagem de $L(G)$ que satisfaz as restrições comportamentais desejadas considerando os eventos não-controláveis do alfabeto de G. Ou seja, $supC(L)$ é a linguagem minimamente restritiva que desconsidera a solução trivial, que é linguagem vazia [RW87a].

Em um contexto de desenvolvimento de sistemas abertos, nos quais os elementos que o compõem comunicam-se através de troca mensagens, a TCS pode ser aplicada para restringir o comportamento de alguns destes elementos inibindo que determinadas mensagens sejam processadas.

Capítulo 3

Reúso e Adaptação

A idéia de reúso de artefatos de software é tida como uma boa abordagem para promover a prática da engenharia de software. Programadores têm reusado idéias, objetos, abstrações e processos para o desenvolvimento de sistemas de software, entretanto as abordagens utilizadas para promover o reúso sempre foram *ad-hoc* [Kre99]. Nos dias atuais, sistemas computacionais maiores e mais complexos precisam ser construídos em períodos de tempo mais curtos e com alto grau de segurança no funcionamento. Isto requer uma abordagem sistematizada para promover o reúso de artefatos de software.

Neste capítulo será discutido o reúso no processo de desenvolvimento de sistemas de software. Serão apresentadas algumas das abordagens para reúso de software e os benefícios promovidos por esta prática. Também será tratado do reúso de modelos formais na construção de sistemas de software, focalizando mais especificamente a atividade de adaptação destes modelos, que é o interesse principal nesta dissertação.

3.1 Abordagens para o Reúso de Software

A Conferência de Engenharia de Software patrocinada pela OTAN[1] em 1968 é, geralmente, considerada o berço da engenharia de software. Nesta conferência discutiu-se

[1] Organização do Tratado do Atlântico Norte.

o problema da construção de sistemas de software complexos, confiáveis e com custos de produção controlados, o chamado problema da *crise de software*. O reúso de software foi apontado como uma possível abordagem para superar este problema. Foi nesta conferência que McIlroy [McI69] propôs a idéia de uma biblioteca de componentes reusáveis e de técnicas automáticas que possibilitassem a adaptação desses componentes a diferentes aplicações [Kru92].

Três décadas após esta conferência, o reúso de software continua sendo visto como uma maneira poderosa de promover práticas de engenharia de software. As vantagens de diminuir o esforço de desenvolvimento de software através de reúso continuam aceitas, ainda que as ferramentas, métodos, linguagens, e boa parte do conhecimento referente à engenharia de software tenha mudado significativamente desde 1968.

Entretanto, as tentativas de reusar software não se tornaram uma prática padrão no processo de desenvolvimento [Kru92]. Com vistas neste fracasso, têm-se renovado o interesse em compreender como e onde o reúso pode ser implementado e quais as causas da dificuldade na implantação da idéia, aparentemente simples, de reúso no processo de desenvolvimento de software [BP89b; BP89b; Fre87; Tra88].

De maneira simplificada, o reúso de software pode ser definido como o reúso de artefatos de software durante a construção de um novo sistema de software. Os tipos de artefatos que podem ser reusados não estão limitados apenas a trechos de código fonte, mas podem incluir decisões de projeto, especificações, documentação, etc.

Krueger [Kru92] estudou e categorizou as diversas abordagens para reúso de software em oito classes: 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. Com base nesse estudo constatou-se que, apesar das diferenças, todas elas apresentam algumas características em comum: *abstração*, *seleção*, *adaptação* (também chamada de especialização) e *integração*.

A abstração é uma característica essencial para qualquer técnica de reúso. Todas

as abordagens de reúso de software envolvem alguma forma de abstração para os artefatos envolvidos. Sem abstrações, os desenvolvedores seriam forçados a examinar minuciosamente uma coleção de artefatos reusáveis na tentativa de identificar o que um determinado artefato faz, onde ele pode ser reusado, e como reusá-lo.

As abordagens de reúso devem possibilitar ao desenvolvedor localizar, comparar, e selecionar artefatos de software reusáveis de forma automatizada. Desta forma, esquemas de classificação devem ser utilizados para organizar uma biblioteca de artefatos reusáveis e guiar os desenvolvedores na busca de artefatos nesta biblioteca.

Após a seleção de um artefato, pode ser desejável que o mesmo seja adaptado através de parâmetros, transformações, restrições, ou alguma forma de refinamento. Por exemplo, uma implementação reusável de uma estrutura de dados do tipo pilha pode ser parametrizada para configurar a profundidade máxima da mesma. Um programador que utilize esta pilha deve adaptá-la provendo um valor para este parâmetro.

Para integrar um artefato reusável em um sistema de software, o projetista deve compreender a interface do artefato, isto é, as propriedades do artefato que interagem com outros artefatos. A interface de um artefato pode ser vista como uma abstração dos detalhes internos do artefato. Além disso, o projetista deve dispor de um arcabouço[2] para integração dos artefatos a serem reusados. O desenvolvedor utiliza o arcabouço de integração para combinar uma coleção de artefatos, selecionados e adaptados, em um sistema de software completo.

Além disso, processos de desenvolvimento de software devem ser utilizados para auxiliar no gerenciamento do projeto do sistema a ser desenvolvido.

De maneira geral alguns destes processos de desenvolvimento, notadamente aqueles baseados em abordagens de ciclo de vida, consideram que o desenvolvimento de um novo sistema de software começa sempre do zero. Desta forma, é preciso adequar tais abordagens para que suas fases incorporem as atividades de reúso. Assim, técnicas automatizadas de reúso de artefatos de software podem ser utilizadas em

[2]Tradução para *framework*.

todo o processo de desenvolvimento.

Em [RBV94] é mostrado um modelo de ciclo de vida de desenvolvimento baseado em reúso de artefatos. Nesse modelo, cada fase do ciclo de vida está relacionado com atividades específicas de reúso. Na Tabela 3.1 são enumeradas as atividades de reúso relacionadas com algumas das fases do modelo de ciclo de vida em cascata [GJM91]. Esta tabela é ilustrativa, e não contempla todas as fases do modelo em questão, servindo apenas para ilustrar a correspondência entre as fases de desenvolvimento e as atividades de reúso. Para maiores detalhes o leitor interessado deve consultar [RBV94].

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

Tabela 3.1: Relação entre fases do modelo cascata e atividades de reúso.

No contexto deste trabalho estamos preocupados com o reúso na etapa de especificação, e os objetos de reúso estudados são as especificações (ou modelos) formais.

Embora à primeira vista possa parecer que apenas a atividade de modelagem seja beneficiada com o estudo aqui apresentado, deve-se advertir que a atividade de encontrar um trecho de código que satisfaça a determinados requisitos será realizada mais facilmente se, primeiramente, for encontrado um modelo formal que represente o comportamento desejado deste código. Desta forma, o desenvolvedor será beneficiado pelas técnicas de reúso de artefatos de software que contemplem a fase de especificação formal durante todo o processo de desenvolvimento.

3.2 Reúso Baseado em Componentes

A crescente demanda por níveis mais elevados de sofisticação nos sistemas de software tem requerido o projeto e a implementação de sistemas cada vez maiores e mais complexos. Juntamente com este fato apresenta-se a necessidade de aumentar a produtividade dos desenvolvedores, a confiança no funcionamento desses sistemas e de promover uma redução nos custos de produção. O desenvolvimento baseado em componentes reusáveis e em arquiteturas de software tem sido utilizado para satisfazer os requisitos expostos acima.

Como dito anteriormente, em 1968 McIlroy [McI69] introduziu a noção de reúso de software propondo uma indústria de componentes de software. Estes componentes deveriam servir como blocos básicos na construção de novos sistemas de software. Desta maneira, pode-se afirmar que um dos objetivos do desenvolvimento baseado em componentes é estabelecer um repositório de componentes reusáveis no qual os desenvolvedores de sistemas possam recuperar esforços de programação anteriores.

A idéia de criar tal repositório é aparentemente simples. Entretanto existem algumas dificuldades a serem superadas. A maior destas dificuldades está relacionada com a necessidade de encontrar boas abstrações para os componentes a serem armazenados [Kru92]. Sem estas abstrações, o usuário do repositório precisará examinar o código fonte de cada componente para determinar o que ele implementa.

Além disso, é necessário prover um mecanismo que possibilite ao desenvolvedor selecionar um componente sem que seja preciso realizar uma busca exaustiva em todo o repositório.

Desta forma, é preciso que os componentes a serem armazenados no repositório sejam empacotados de maneira a permitir sua distribuição e composição com outros componentes. Um componente pode ser definido da seguinte maneira:

> "Um componente é uma unidade de composição de aplicações de software, que possui um conjunto de interfaces e um conjunto de requisitos, e que deve poder ser desenvolvido, incorporado e composto com outros

componentes de forma independente, em tempo e espaço" [Szy98].

As interfaces de um componente determinam tanto as operações que o componente implementa como aquelas que este precisa utilizar de outros componentes durante sua execução. Os requisitos determinam os recursos que um componente necessita.

Uma vez estabelecido o conceito de componente, um modelo de componentes define a forma de suas interfaces e os mecanismos utilizados para interconectá-los. Baseada em um modelo de componentes concreto, uma plataforma de componentes é um ambiente de desenvolvimento e execução de componentes que permite isolar a maior parte das dificuldades conceituais e técnicas existentes na construção de aplicações baseadas nesse modelo. Desta forma, pode-se definir uma plataforma como uma implementação dos mecanismos do modelo, junto com uma série de ferramentas associadas.

Entretanto, apenas dispor de componentes de software não é suficiente para desenvolver aplicações. Um outro aspecto importante na construção de sistemas complexos está relacionado com o projeto da estrutura do sistema, as chamadas Arquiteturas de Software.

Entende-se por Arquitetura de Software a representação de alto nível da estrutura de um sistema ou aplicação, que descreve as partes que o integram, as interações entre estas partes, os padrões de composição utilizados, e as restrições de aplicação desses padrões. A descrição desta estrutura representa grandes esforços de projeto e implementação que podem ser reusados acarretando em benefícios significativos no desenvolvimento de sistemas de software. No contexto deste trabalho não estamos preocupados com este aspecto, mas sim como tais esforços podem ser adaptados. O leitor interessado deve consultar [SG96] para obter maiores detalhes.

3.3 Reúso de Modelos Formais

O reúso de artefatos de software não está restrito ao reúso de código fonte. De maneira semelhante à que ocorre com os componentes, modelos formais também podem ser projetados objetivando seu posterior reúso.

Construir um modelo formal de um sistema em desenvolvimento, durante a fase de projeto, pode tornar-se uma tarefa mais produtiva caso o projetista disponha de pedaços de modelos construídos anteriormente que possam ser reusados. Devido ao fato dos modelos formais serem descritos através de uma linguagem matemática é possível investigar, e alterar, seu comportamento de maneira automática. Desta forma, as tarefas de recuperação e de adaptação de artefatos de software podem ser realizadas mais facilmente.

As vantagens de reusar esforços de modelagem são bastante evidentes para qualquer projetista que já tenha reaproveitado os seus próprios esforços, e dispor de um mecanismo automático que auxilie nesta tarefa pode tornar a atividade ainda mais produtiva.

Diversos trabalhos que sugerem a adoção das atividades de reúso de software durante a modelagem formal de sistemas de software já foram propostos. Alguns deles sugerem a adoção de conceitos e mecanismos de orientação a objetos, tais como herança e encapsulamento, em redes de Petri [Lak95; vdAB96; LHCB98; Gue97]. Outros sugerem o desenvolvimento de um sistema de padrões de redes de Petri [NJ98] de maneira semelhante aos padrões de projeto, generalizando modelos em redes de Petri para reúso posterior.

Entretanto, todos os trabalhos acima mencionados promovem o reúso de modelos através de mecanismos de herança, o que implica que o projetista deve deter algum conhecimento sobre os detalhes de implementação do artefato com qual está lidando (reúso "caixa-branca"). No âmbito deste trabalho, utiliza-se a idéia de reúso "caixa-preta", pois apenas informações referentes ao comportamento do modelo precisam, *a priori*, ser de conhecimento do projetista.

No contexto de reúso de modelos os sistemas não são modelados a partir do

zero, mas sim a partir de outros modelos já existentes. Desta forma, é preciso que o projetista passe a se preocupar em como, e onde, buscar e adaptar pedaços de modelos que possam ser reusados na construção do modelo sendo projetado. Além disso, ele deve procurar identificar potenciais candidatos ao reúso, para que os mesmos sejam armazenados em um repositório.

Desta forma identificam-se as seguintes atividades de reúso durante a modelagem formal de sistemas de software:

1. Identificar as partes do modelo do novo sistema;
2. Selecionar as partes que necessitam ser totalmente construídas e aquelas que podem ser objeto de reúso;
3. Descrever e recuperar os objetos de reúso;
4. Adaptar os modelos (partes) recuperados ao projeto em desenvolvimento;
5. Integrar os modelos recuperados/adaptados;
6. Identificar (sub-)modelos reusáveis e armazená-los no repositório.

Em um contexto informal, em que a busca por um artefato é feita através da verificação de sua assinatura, o processo de recuperação pode ser frustrante. Pois podem ser recuperados diversos artefatos que apesar de atenderem as restrições especificadas (assinatura), não satisfazem a um comportamento desejado. Por exemplo, na busca de uma função (armazenada em uma biblioteca de funções) que receba como parâmetro dois inteiros e retorne um inteiro igual a soma dos valores recebidos, um método de recuperação que leve em consideração apenas a assinatura (parâmetros + tipo de retorno) poderá retornar funções que implementem a soma de inteiros, a subtração, a multiplicação, a divisão, entre outras.

Neste exemplo, não há garantias de encontrar um artefato adequado pelo fato de uma informação puramente sintática (assinatura) ser insuficiente para descrever o que ele faz. Por outro lado, se a recuperação dos artefatos for feita a partir de

restrições de comportamento (informação semântica), pode-se chegar ao caso de que a descrição do artefato seja igual à sua implementação (no caso, modelo a ser recuperado), e neste caso deixa de ter sentido a atividade de reúso.

Devido ao fato de que as linguagens de especificação formal orientadas a modelos (redes de Petri, por exemplo) não são adequadas para a descrição de propriedades, é necessário adotar linguagens que permitam descrever o comportamento dos modelos que deverão ser armazenados que situem-se entre entre os dois extremos (sintático $\times$ semântico). Desta forma, além de obter-se uma maneira apropriada para a descrição das propriedades desejadas dos modelos, evita-se a situação na qual a descrição do modelo seria o próprio modelo.

É preciso, ainda, prestar atenção ao fato de que quanto mais restrito for o critério de busca (mais próximo do nível semântico), mais o projetista descreve o modelo desejado. De outra forma quanto menos restritos forem os critérios mais alterações serão necessárias no modelo recuperado para integrá-lo ao sistema em desenvolvimento.

Ainda assim, seria bastante ingênuo assumir que o mecanismo de recuperação automática de modelos sempre encontrará algum que irá satisfazer completamente às restrições desejadas, até mesmo pelo fato do custo computacional para fazê-lo ser proibitivo. É bastante razoável admitir que o modelo recuperado requererá alguma adaptação. Desta forma, torna-se necessária a utilização de um mecanismo que dê suporte à adaptação destes modelos.

A seguir será discutida uma solução para o reúso de modelos formais tendo como foco principal a atividade de adaptação, que é o objeto de estudo deste trabalho.

3.4 Adaptação de Modelos para Reúso

Algumas práticas de projetos de engenharias bem estabelecidas consistem em definir construções básicas a partir dos quais sistemas complexos serão construídos. Estas construções são representados através de modelos matemáticos que podem ser ana-

lisados, verificados e/ou simulados com o objetivo de garantir que o comportamento desta construção satisfaz a um conjunto de requisitos. Além disso, investigam-se mecanismos que permitam armazená-los adequadamente, recuperá-los quando necessário, adaptá-los aos requisitos do projeto e integrá-los ao projeto.

No contexto deste trabalho as construções básicas são descritos através de modelos em redes de Petri Coloridas [Jen92], utilizando a ferramenta Design/CPN. O reúso destes modelos pode ser aplicado satisfatoriamente se existir um conjunto de técnicas e ferramentas que dêem suporte ao processo de reúso. Desta forma, neste trabalho será introduzida uma solução para a adaptação de modelos formais descritos em redes de Petri Coloridas.

A abordagem para adaptação de modelos descrita neste trabalho é inspirada no conceito de adaptação de modelos, e baseia-se nos resultados da Teoria do Controle Supervisório e na técnica de Verificação Automática de Modelos.

Como dito no Capítulo 2 um dos principais resultados da TCS é a definição do algoritmo da suprema sub-linguagem controlável ($supC(L)$). Dada a especificação de um sistema e um conjunto de restrições de comportamento, a aplicação do algoritmo resulta em uma nova especificação que satisfaz às restrições. De fato esta nova especificação, no contexto deste trabalho, é vista como o modelo adaptado, ou seja, o modelo a ser reusado.

Entretanto, é importante observar que o algoritmo da $supC(L)$ opera sobre o espaço de estado do modelo do sistema. Geralmente, a explosão do tamanho do espaço de estados dos modelos é o maior problema encontrado quando da aplicação deste algoritmo. Assim, é necessário investigar mecanismos que permitam representar e manipular eficientemente este espaço de estados.

Algumas diretrizes básicas norteiam a abordagem para adaptação de modelos introduzida neste trabalho. A seguir essas diretrizes serão enumeradas.

1. É preciso dispor de uma linguagem adequada à descrição de modelos formais, preferivelmente que possua uma ferramenta de edição análise e associada. Neste trabalho optou-se por utilizar redes de Petri Coloridas para descrever os

modelos, tendo como suporte a ferramenta Design/CPN;

2. Além disso é preciso dispor de uma linguagem adequada à descrição das propriedades desejadas dos modelos;

3. Verificar se o comportamento do modelo, que no caso das redes de Petri é dado pelo seu grafo de alcançabilidade (espaço de estados), satisfaz às propriedades descritas;

4. Modificar o modelo, se for possível derivar o comportamento desejado. Em outras palavras, deve-se alterar o modelo para inibir a ocorrência de alguma transição quando o estado alcançado não respeitar às restrições de comportamento estabelecidas;

5. Finalmente, deve-se exportar o modelo adaptado para que seja integrado ao sistema em desenvolvimento.

É importante ressaltar que o armazenamento e a recuperação de modelos é um aspecto importante no contexto de reúso, mas foge do escopo deste trabalho. O leitor interessado em obter informações sobre como armazenar e recuperar modelos do repositório de modelos em redes de Petri Coloridas pode consultar [Lem01].

Com base nessas diretrizes detalha-se no Capítulo 4 a abordagem para a adaptação de modelos reusáveis introduzida neste trabalho.

Capítulo 4

Uma Solução para Adaptação de Modelos

Neste capítulo são apresentados os principais resultados deste trabalho. Na Seção 4.1 será introduzida uma abordagem para a adaptação de modelos. Na Seção 4.2 será apresentado o procedimento que implementa a metodologia proposta, e por fim na Seção 4.3 será introduzido um exemplo de aplicação da metodologia.

4.1 Descrição da Abordagem

Como estabelecido anteriormente, o objetivo principal deste trabalho é definir uma abordagem que, a partir de uma especificação formal em redes de Petri e de um conjunto de restrições de comportamento descritas em lógica temporal, resulte em uma nova especificação que atenda às restrições desejadas.

Como dito no Capítulo 3, uma forma de atingir este objetivo é aplicar diretamente o algoritmo da suprema sub-linguagem controlável ($supC(L)$). Entretanto, caso esta linguagem seja vazia, isto só será detectado no final da execução do algoritmo, o que pode ter um alto custo computacional.

Portanto, as especificações são primeiramente validadas contra o espaço de estados do modelo de rede de Petri, e só então restringe-se o seu comportamento. Neste

trabalho, adota-se uma abordagem utilizando a técnica de verificação automática de modelos que busca solucionar este problema.

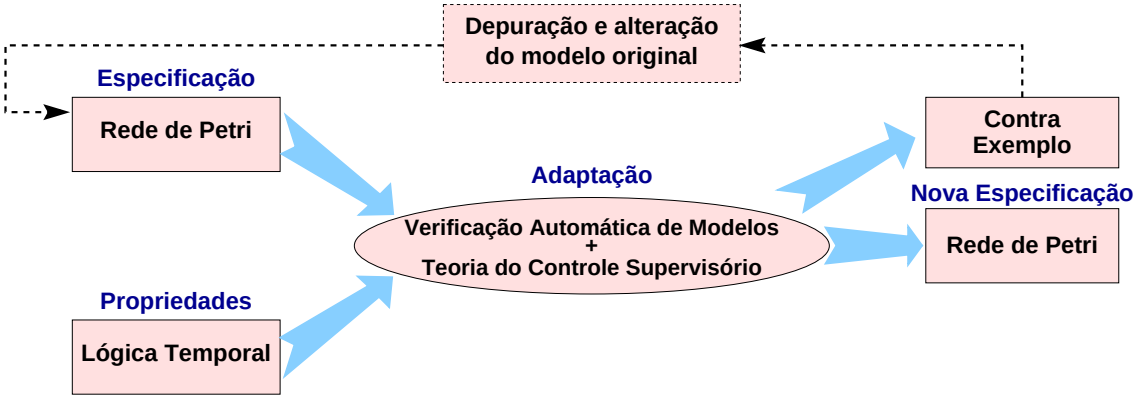

Figura 4.1: Abordagem para adaptação de modelos reusáveis.

Na Figura 4.1 é ilustrada a abordagem aqui introduzida. O primeiro passo para realizar a adaptação de um modelo para que o mesmo satisfaça às restrições definidas é a geração do grafo de ocorrência (espaço de estados) da rede de Petri (modelo) fornecida inicialmente. Uma vez obtido este grafo, deve-se verificar as restrições.

O funcionamento dos verificadores automáticos de modelos consiste em retornar uma mensagem afirmativa, caso a propriedade sendo verificada seja satisfeita, ou um contra-exemplo caso contrário.

Uma vez realizada a verificação automática sobre o modelo fornecido, o projetista irá dispor, além da rede de Petri fornecida, de um conjunto de restrições que devem ser aplicadas sobre o comportamento dessa rede e, além disso, saberá se é possível obter um controlador que restrinja o comportamento da rede de Petri de forma a satisfazer as propriedades desejadas.

Caso não seja possível realizar a síntese automaticamente a partir do modelo fornecido, é necessário que o projetista do sistema altere o modelo fornecido de forma a tornar a síntese viável ou então que forneça um novo modelo.

Caso a síntese seja possível, deve-se aplicar o algoritmo de síntese de supervisores sobre o grafo de ocorrência, e a medida que estados indesejáveis são encontrados, os mesmo são marcados como tal para que depois seja possível alterar a rede de

Petri fornecida de forma a impedir que tais estados sejam atingidos. Dessa forma será obtida uma lista contendo todas as marcações (estados) do grafo de ocorrência consideradas indesejáveis.

Com base nesta lista, modifica-se a rede de Petri fornecida de modo a satisfazer as restrições.

Para obter este resultado deve-se induzir (controlar) o comportamento da rede de maneira a evitar que os estados marcados como indesejáveis sejam atingidos. Uma maneira de fazê-lo é inibir a ocorrência de determinadas transições, quando o disparo delas leve a um estado indesejável.

Para tornar esta ação possível deve-se enumerar todos os nós do grafo de ocorrência gerado anteriormente. Além disso, é preciso adicionar à rede novos lugares, denominados lugares de controle, cujas marcações iniciais corresponderão ao rótulo dado à marcação inicial do grafo marcado. Estes novos lugares deverão ser lugares de entrada e saída das transições da rede, e sempre que alguma dessas transições ocorrer, será executado um trecho de código (gerado automaticamente) que determinará o valor (que corresponderá ao rótulo da próxima marcação alcançada) da ficha que será colocada nestes lugares de controles. Desta forma, a informação contida na ficha deverá refletir sempre os rótulos existentes no grafo marcado, ou seja, a marcação atual da rede.

Uma vez assegurado que o valor das fichas contidas nos lugares adicionados à rede representa a marcação em que a rede de Petri sem restrições de controle teria atingido para uma dada seqüência de disparos, é preciso impedir que os estados marcados como indesejáveis sejam atingidos. Para fazê-lo é preciso inspecionar cada transição do modelo e averiguar se a sua ocorrência pode, em algum instante, levar a um estado indesejável. Caso seja possível, e a transição seja tida como controlável, deve-se adicionar a esta transição uma guarda inibindo o seu disparo nos estados em que o mesmo leva a um estado não-desejável.

4.2 Implementação do Procedimento

Dada uma rede de Petri Colorida não controlada RP_{NC} definida de acordo com [Jen92], e um conjunto de restrições de comportamento descritas em ASK-CTL, a seguir, será introduzido um procedimento para adaptar RP_{NC} de maneira a obter uma nova rede de Petri Colorida RP_C cujo comportamento deverá satisfazer as restrições desejadas.

A execução do algoritmo de síntese de supervisores sobre o espaço de estados de RP_{NC} gera como uma resultado uma lista contendo as marcações indesejáveis. Com base nesta lista, modifica-se RP_{NC} para evitar que os estados indesejados sejam alcançados da seguinte maneira:

- Adicionar ao conjunto de lugares $P \in RP_{NC}$ lugares de controle ($Lugar\_de\_Controle_i$) com marcação inicial correspondente ao rótulo da marcação inicial do grafo gerado anteriormente.

- Também será necessário adicionar ao nó de declaração global da rede de Petri fornecida inicialmente uma função de controle (chamada de $Funcao\_de\_Controle$) que deverá receber como parâmetro o identificador de uma transição e o rótulo de uma marcação e retornará, com base nos parâmetros recebidos, o rótulo da próxima marcação a ser atingida. $Funcao\_de\_Controle$ deverá ser construída com base no grafo de ocorrência rotulado construído anteriormente. Baseado nas informações contidas no grafo (rótulo dos nós de origem e destino de cada arco e identificador da transição disparada) é possível construir tal função.

- Finalmente, é preciso controlar o disparo das transições inibindo-os sempre que rede possa atingir um estado indesejável. Como foi estabelecido anteriormente isto pode ser feito adicionando guardas às transições. Desta forma tem-se que sobre cada transição $t \in T$ cuja ocorrência pode ocasionar em uma violação nas restrições de comportamento deve-se realizar as seguintes operações:

1: **para cada** $Lugar\_de\_Controle_i$ **faça**
2: Criar arco de $Lugar\_de\_Controle_i$ para t;
3: Adicionar código à transição t com uma chamada à função $Funcao\_de\_Controle$ passando como parâmetros o identificador de t e o rótulo da marcação atual da rede, dada pelo valor da ficha em $Lugar\_de\_Controle_i$;
4: Criar arco de t para $Lugar\_de\_Controle_i$ cuja inscrição será a variável de saída do código adicionado à t;
5: **se** Se t for controlável **então**
6: **para cada** Marcação M pertencente ao grafo de ocorrência de RP_{NC} considerada desejável **faça**
7: **se** Próxima marcação alcançável a partir de M for indesejável **então**
8: Adicionar guarda à t inibindo seu disparo quando valor da ficha em $Lugar\_de\_Controle_i$ for equivalente ao rótulo de M
9: **fim se**
10: **fim para**
11: **fim se**
12: **fim para**

Deve-se observar que o disparo de uma transição não-controlável poderá atualizar o valor da ficha em $Lugar\_de\_Controle$. O fato de se detectar o disparo de uma transição não implica, necessariamente, em poder controlá-la. Note que as transições são inibidas ou habilitadas pelas suas guardas juntamente com as informações das fichas em $Lugar\_de\_Controle_n$.

4.3 Exemplo de Aplicação do Procedimento

Para ilustrar o procedimento de adaptação de modelos introduzido, apresenta-se um exemplo de sua aplicação destacando os principais passos envolvidos.

Na Figura 4.2 é mostrada uma rede de Petri que é um modelo para um sistema ferroviário composto por seis secções (representadas pelos lugares $S[1...6]$) por onde circulam dois trens. Todas as passagens entre uma secção e outra (transições da rede) podem ser controladas, excetuando-se a passagem da secção 6 para a 1 (transição $T6\_to\_1$). Inicialmente há dois trens no sistema que se encontram nas seções $S1$ e $S4$. Na Figura 4.2 apresenta-se o nó de declaração global para esta rede.

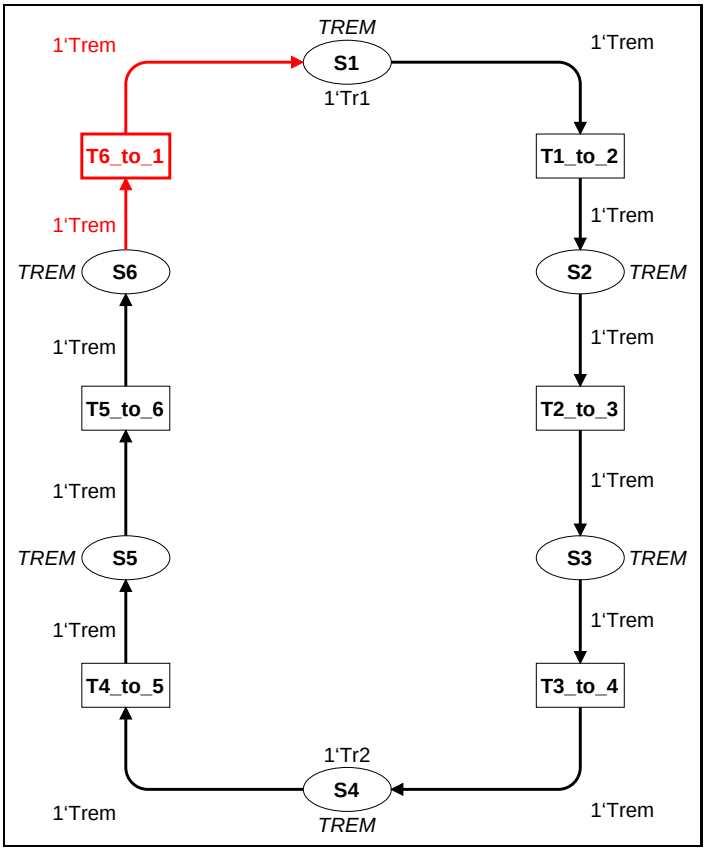

Figura 4.2: Sistema ferroviário sem restrições de controle.

Por questões de segurança é preciso que nunca dois trens do sistema ocupem, em algum instante de tempo, a mesma secção ou secções adjacentes. Entretanto, o modelo da Figura 4.2 não atende às restrições definidas, e é preciso modificá-lo de

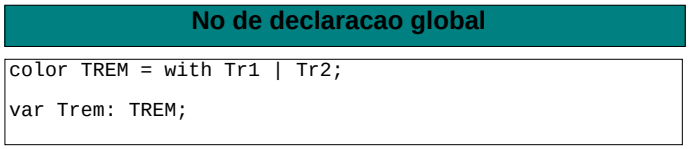

Figura 4.3: Nó de declarações globais para o sistema férreo sem controle.

modo a satisfazê-las. Isto é conseguido através da síntese do supervisor, que, de fato como dito no Capítulo 3, é o modelo a ser reusado.

Para construir este supervisor será utilizado o procedimento descrito na Seção 4.2. Para fazê-lo, o primeiro passo é descrever as propriedades desejadas. Ou seja, neste caso as propriedades são:

- "É possível que nunca dois trens ocupem a mesma secção"
  $POS(NOT(NF(\text{"Dois trens na mesma secção"}, MesmaSeccao)))$
- "É possível que nunca dois trens ocupem secções adjacentes"
  $POS(NOT(NF(\text{"Dois trens em secções adjacentes"}, SeccoesAdjacentes)))$

Nas fórmulas escritas utilizando ASK-CTL, NF é uma função do ASK-CTL utilizada para descrever fórmulas de estados, e tem como parâmetros um identificador e uma função (que deve ser escrita pelo projetista) que deverá receber como parâmetro um nó do espaço de estados e retornar um valor booleano. Esta função descreve a propriedade que será avaliada em cada marcação do grafo de ocorrência da rede. NOT é o operador de negação. E, por fim, POS é um quantificador existencial de caminhos.

Verificando as fórmulas ASK-CTL constrói-se então um supervisor a partir do modelo fornecido.

Uma vez verificada a existência do supervisor, deve-se construir uma lista com todos os estados indesejáveis. Assim, o comportamento desejado do modelo será dado pelo o grafo de ocorrência excetuando-se os estados presentes nesta lista. No exemplo apresentado o comportamento desejado do modelo é representado pelo grafo mostrado na Figura 4.4. Neste grafo os nós com as bordas pontilhadas são as

marcações da rede que não devem ser alcançadas. E os arcos pontilhados indicam as ocorrências da transição não-controlável ($T6\_to\_1$). O leitor deve observar que os nós de números 9 e 35, apesar de representarem marcações válidas no sistema, devem ser evitados, pois uma vez alcançados não será possível impedir que o sistema alcance um estado indesejável.

Para controlar o comportamento do modelo é preciso inibir a ocorrência de qualquer elemento de ligação que leve a um estado indesejado. E isso é feito adicionando-se guardas às transições que inibem o seu disparo quando podem levar a estados indesejáveis.

Na Figura 4.5 pode-se observar que foi criado um lugar (chamado $Controle1$) do qual todas as transições retiram e colocam fichas. As fichas contidas neste lugar armazenam informações sobre o estado da rede, e é com base no valor dessas fichas que as guardas das transições serão avaliadas. Desta forma é possível inibir o disparo de uma transição sempre que preciso. Além disso, também observa-se que existem trechos de códigos associados a cada transição. Esses códigos são executados sempre que a transição ao qual está associado dispara, e sua função é "descobrir" qual será a próxima marcação alcançada e manter a informação (fichas) em $Controle1$ sempre atualizada.

Na Figura 4.6 é apresentado o nó de declaração global modificado contendo as declarações das cores e variáveis utilizadas em $Controle1$ e em seus respectivos arcos, e a função $Funcao\_de\_Controle$ que determinará antecipadamente qual a próxima marcação a ser atingida.

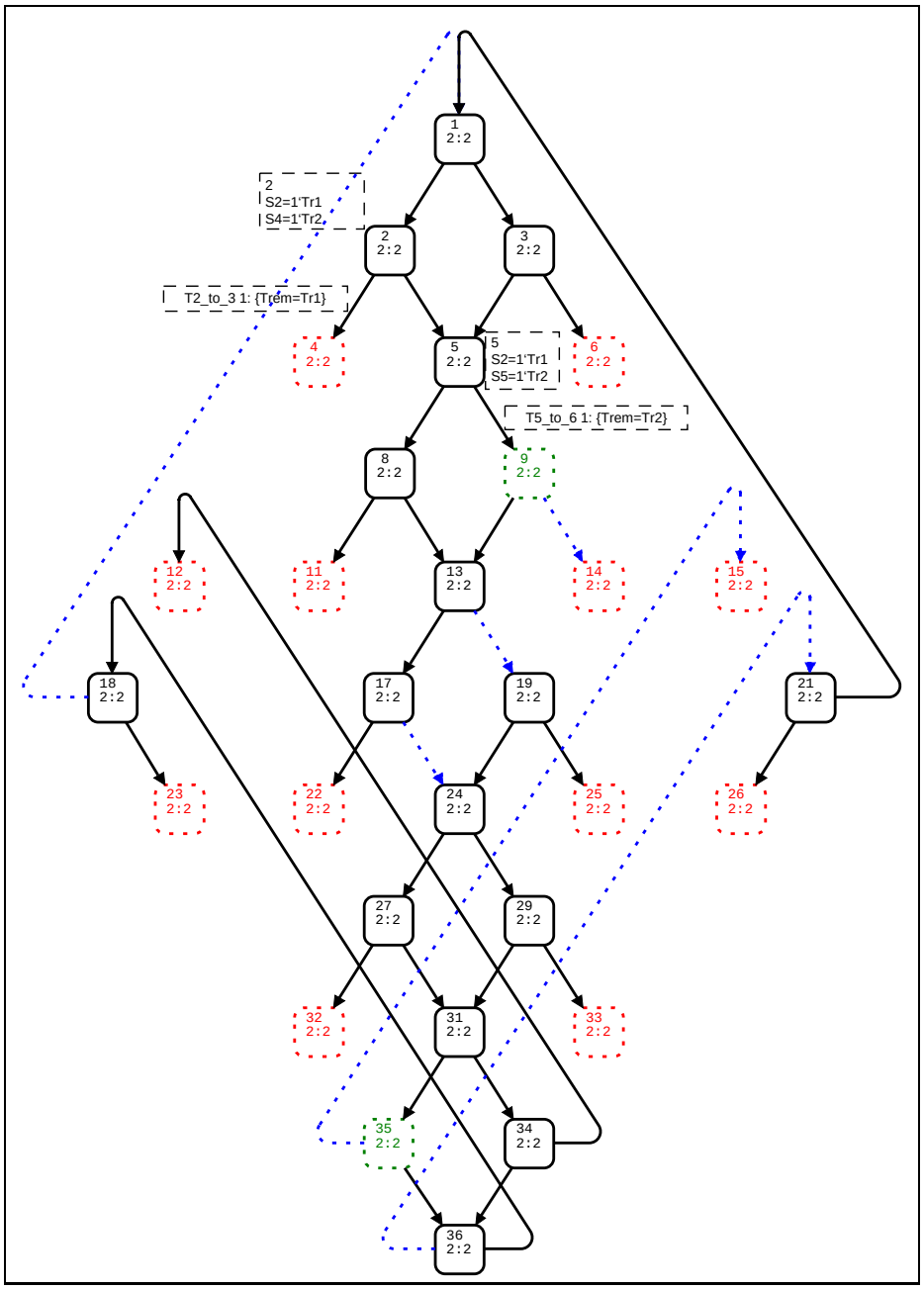

Figura 4.4: Grafo de ocorrência do sistema controlado.

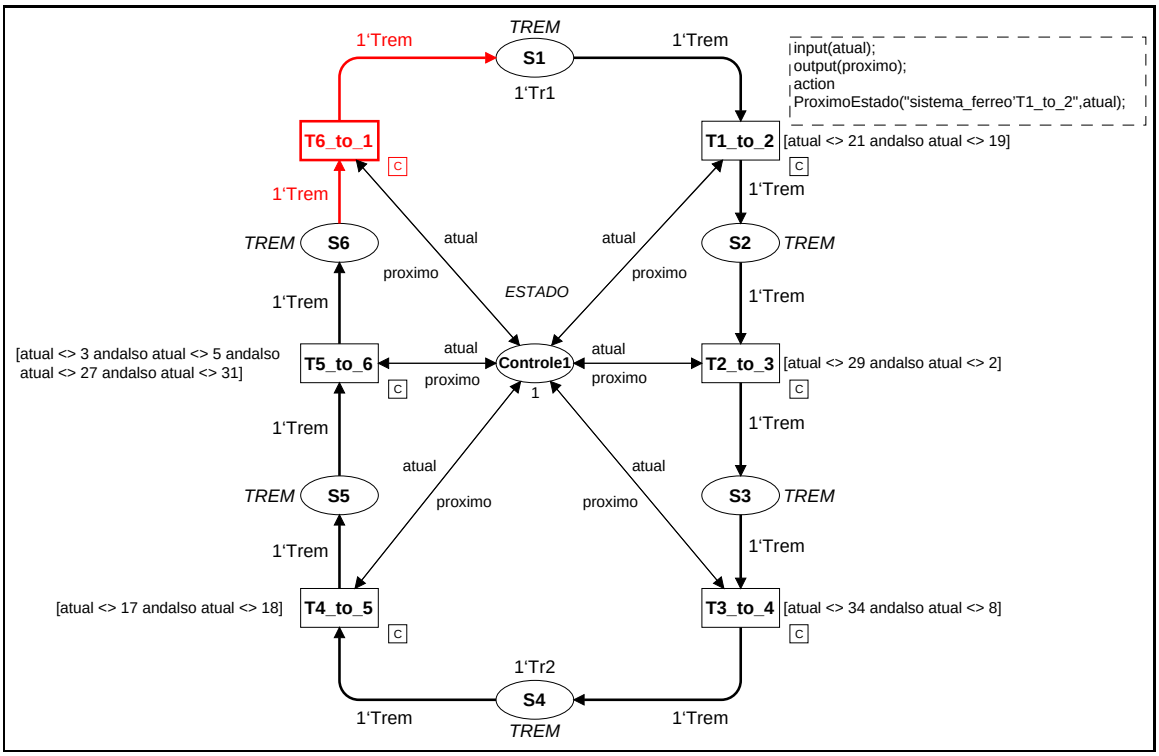

Figura 4.5: Sistema ferroviário com restrições de controle.

No de declaracao global

```
color TREM = with Tr1 | Tr2;

var Trem: TREM;


(*
 * Funcao criada para determinar durante a simulacao
 * qual o proximo estado a ser alcanado
 *)


color ESTADO = int;
var atual: ESTADO;
var proximo: ESTADO;

fun Funcao_de_Controle(transicao, marcacao) =
 case (transicao, marcacao) of
  ("sistema_ferreo'T1_to_2",1) => 2  |
  ("sistema_ferreo'T1_to_2",3) => 5  |
  ("sistema_ferreo'T1_to_2",24) => 29  |
  ("sistema_ferreo'T1_to_2",27) => 31  |
  ("sistema_ferreo'T2_to_3",5) => 8  |
  ("sistema_ferreo'T2_to_3",31) => 34  |
  ("sistema_ferreo'T3_to_4",13) => 17  |
  ("sistema_ferreo'T3_to_4",19) => 24  |
  ("sistema_ferreo'T3_to_4",21) => 1  |
  ("sistema_ferreo'T3_to_4",36) => 18  |
  ("sistema_ferreo'T4_to_5",24) => 27  |
  ("sistema_ferreo'T4_to_5",29) => 31  |
  ("sistema_ferreo'T4_to_5",1) => 3  |
  ("sistema_ferreo'T4_to_5",2) => 5  |
  ("sistema_ferreo'T5_to_6",8) => 13  |
  ("sistema_ferreo'T5_to_6",34) => 36  |
  ("sistema_ferreo'T6_to_1",13) => 19  |
  ("sistema_ferreo'T6_to_1",17) => 24  |
  ("sistema_ferreo'T6_to_1",18) => 1  |
  ("sistema_ferreo'T6_to_1",36) => 21
;

(*
 * Fim da funcao de controle
 *)
```

Figura 4.6: Nó de declaração global com informações de controle.

Capítulo 5

Um Estudo de Caso de Adaptação

No Capítulo 4 apresentou-se a técnica de adaptação de modelos para redes de Petri coloridas introduzida nesse trabalho e ilustrou-se a sua aplicação. Neste Capítulo a técnica será aplicada num contexto de reúso de modelos.

5.1 Um Cenário de Reúso de Modelos

Como dito anteriormente, no contexto de reúso de modelos, há quatro aspectos relevantes: abstração, recuperação, adaptação e integração. Apesar do foco deste trabalho ser o problema da adaptação, é importante vislumbrar os outros aspectos. Para tanto, como introduzido em [Lem01], considere a situação em que é modelado um sistema de controle de tráfego ferroviário. Um sistema desses é composto de muitos módulos, e um desses módulos relaciona-se com a necessidade das locomotivas realizarem manobras com o objetivo de efetuar a troca de vagões, alterando a composição do trem.

Num modelo que represente este módulo, além das partes que representam a configuração da linha férrea e suas conexões, existem partes que controlam as mudanças de composições.

Na Figura 5.1 é ilustrada a chegada de um trem com três componentes (uma locomotiva e dois vagões) ao sistema de troca de composições e sua partida apenas

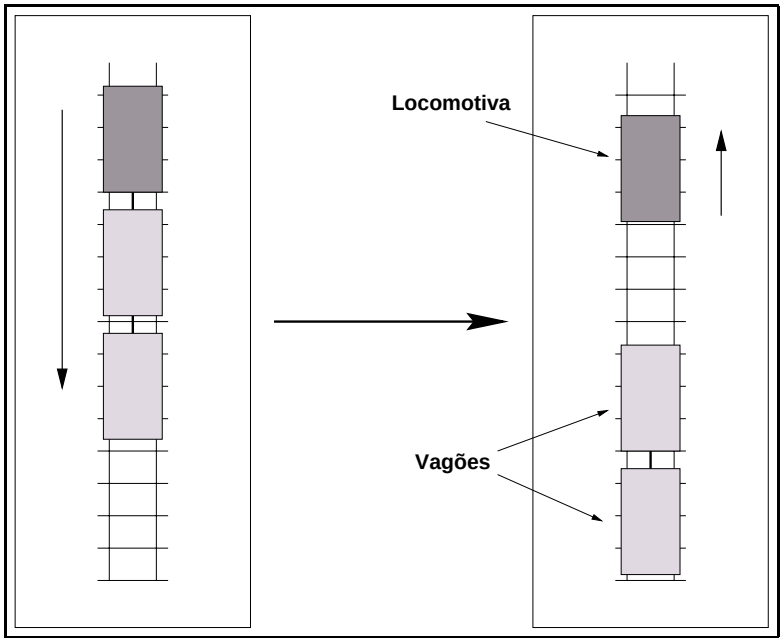

Figura 5.1: Sistema de troca de composições.

com a locomotiva. De maneira semelhante à ilustrada na Figura 5.1 é possível que um outro trem chegue ao sistema de troca de composições deixando mais vagões ou retirando os que lá foram deixados. Um projetista que esteja modelando este sistema notará facilmente que ele pode ser modelado através de uma estrutura de dados do tipo pilha [Knu68] em que os objetos a serem manipulados representam os vagões dos trens.

De maneira semelhante à ilustrada na Figura 5.1 pode haver situações em que o modelo obtido seja uma estrutura de dados do tipo fila. Por exemplo, sempre que um trem entrar em uma determinada secção do sistema férreo seguido, à uma distância segura, por um outro trem é natural que aquele eles saiam desta secção na mesma ordem em que entraram.

Os modelos da situação descrita acima são fortes candidatos ao reúso em contextos diferentes. Podendo promover uma diminuição nos esforços de modelagem em projetos ainda por vir. Assim, o projetista destes modelos deve armazená-los de maneira que sua posterior recuperação possa ser feita de maneira sistemática.

Em uma outra situação, um projetista responsável pela modelagem de parte de

um sistema flexível de manufatura. Neste sistema ele deve modelar o empilhamento e o desempilhamento de matéria prima (e processada) nos depósitos de entrada e saída das células de produção [dCB00; dSR99] de acordo com o ilustrado na Figura 5.2. Além disso, deve modelar o transporte de material entre essas células através de esteiras rolantes. Novamente este projetista depara-se com uma situação em que é necessária realizar a modelagem de estruturas de dados do tipo pilha e do tipo fila.

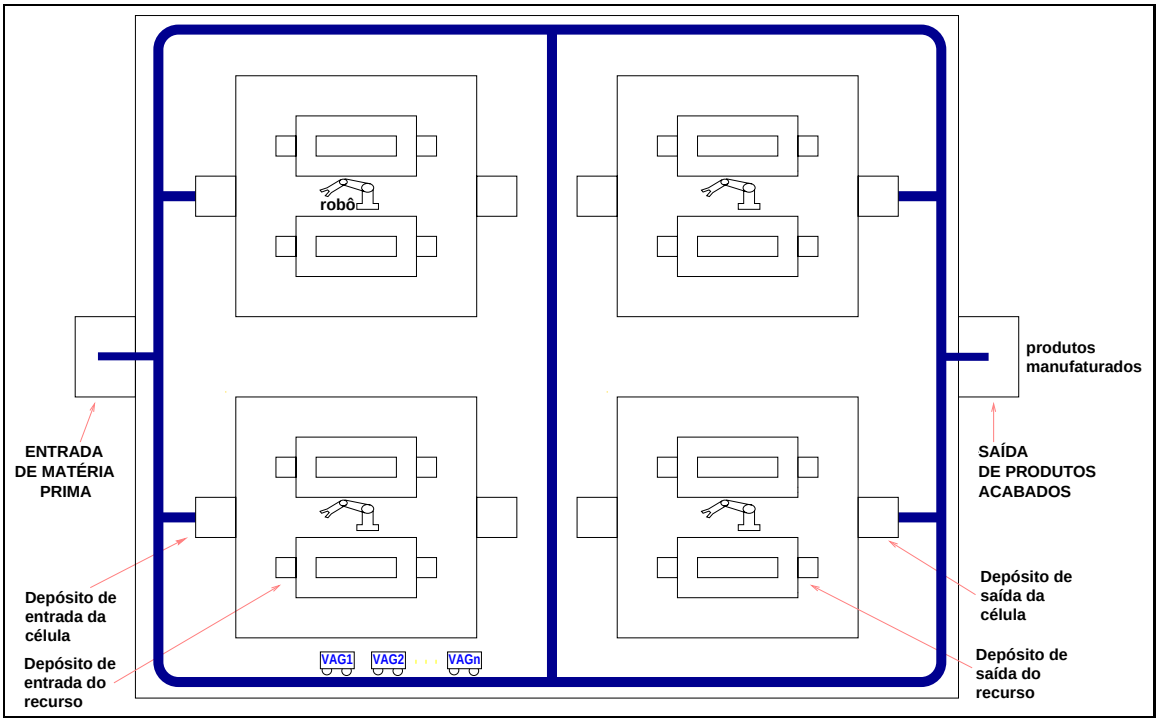

Figura 5.2: Sistema flexível de manufatura com 4 células de produção.

Entretanto, neste caso, o projetista não mais deverá construir esse modelos a partir do zero. Agora ele pode recorrer ao repositório de modelos em busca daqueles que foram adicionados na situação anterior para reaproveitá-los. Admitindo que dispõe-se de um método automático para recuperar esses modelos, e que esse método sempre encontra modelos que satisfaçam aos critério desejados, o projetista irá economizar um tempo considerável na atividade de modelagem uma vez que os modelos de que necessita já estão prontos.

Deve-se ainda acrescentar que sempre que um novo projeto é concluído, ou durante seu desenvolvimento, o projetista responsável deve estar atento para a identi-

ficação de modelos potencialmente reusáveis com o intuito de armazená-los.

5.1.1 Armazenando Modelos

De acordo com [Lem01] o processo de armazenamento de modelos em redes de Petri Coloridas pode ser dividido em quatro etapas. A seguir estas etapas são apresentadas sucintamente:

1. Criação do repositório, caso este ainda não exista. Utilizando a ferramenta Design/CPN deve-se criar uma nova página que conterá o índice do repositório. Nesta página serão adicionadas transições de substituição que levarão à diferentes domínio de aplicação. Sempre que se deseje adicionar um novo domínio deve-se adicionar uma transição de substituição;

2. Após a criação do índice do repositório deve-se criar o nó de declaração global. Devem ser definidos as cores dos lugares da página de índice e dos domínios de aplicação;

3. Depois deve-se criar a página correspondente ao domínio de aplicação recém adicionado. Nesta página deve existir um ambiente de uso para cada modelo do domínio que será armazenado. Neste ambiente existirá uma outra transição de substituição que levará ao modelo que deseja-se armazenar/recuperar;

4. Finalmente, deve-se criar mais uma página que conterá o modelo a ser armazenado e eventualmente recuperado.

5.1.2 Recuperando Modelos

Apenas possuir os modelos armazenados não é suficiente para promover o reúso destes. É preciso dispor de mecanismos que possibilitem sua recuperação. [Lem01] define cinco etapas para a recuperação dos modelos armazenados no repositório.

1. Primeiramente deve-se executar o Design/CPN, carregar o modelo do repositório e ativar a ferramenta de manipulação do grafo de ocorrência;

2. Depois é preciso descrever o comportamento do modelo que se deseja recuperar. Esta descrição é feita utilizando a lógica temporal ASK-CTL;

3. Deve-se iniciar o procedimento de busca. Uma vez iniciado, o projetista deverá identificar o domínio de aplicação em que se deseja efetuar a busca;

4. Cada modelo do domínio escolhido é automaticamente verificado. A medida que modelos que satisfaçam as restrições de desejadas forem sendo encontrados o projetista é informado;

5. O modelo é exportado.

Observe que os passos quatro e cinco ocorrem simultaneamente. A medida que modelos são encontrados é solicitado ao projetista para informar se o modelo deve ser exportado e em qual arquivo deseja-se fazê-lo.

5.2 Adaptando Modelos

No contexto de reúso de modelos o armazenamento e a recuperação de modelos são aspectos fundamentais. Entretanto é importante observar que pode haver a necessidade de adaptar modelos recuperados através da restrição de seu comportamento. Então, consideramos nessa secção a aplicação da técnica de adaptação introduzida no Capítulo 4 para sintetizar modelos a partir de um conjunto de restrições de comportamento. Buscando, desta forma, disponibilizar um meio automático para modificação de modelos recuperados, evitando assim que o projetista precise modificar o componente manualmente. Observe que isto é fundamental para evitar introdução de erros durante o processo de modificação.

Retornando ao cenário descrito na Seção 5.1, o projetista do sistema ferroviário identificou os modelos de pilha e fila como reusáveis e armazenou-os no repositório como descrito na Seção 5.1.1. Posteriormente, quando o mesmo projetista deparou-se com a necessidade de modelar o sistema de transporte entre células de produção do sistema flexível de manufatura, ele fez uma pesquisa no repositório e recuperou,

como descrito na Seção 5.1.2, os modelos da pilha e da fila construídos anteriormente. Os modelos recuperados são mostrados nas Figuras 5.3 e 5.4.

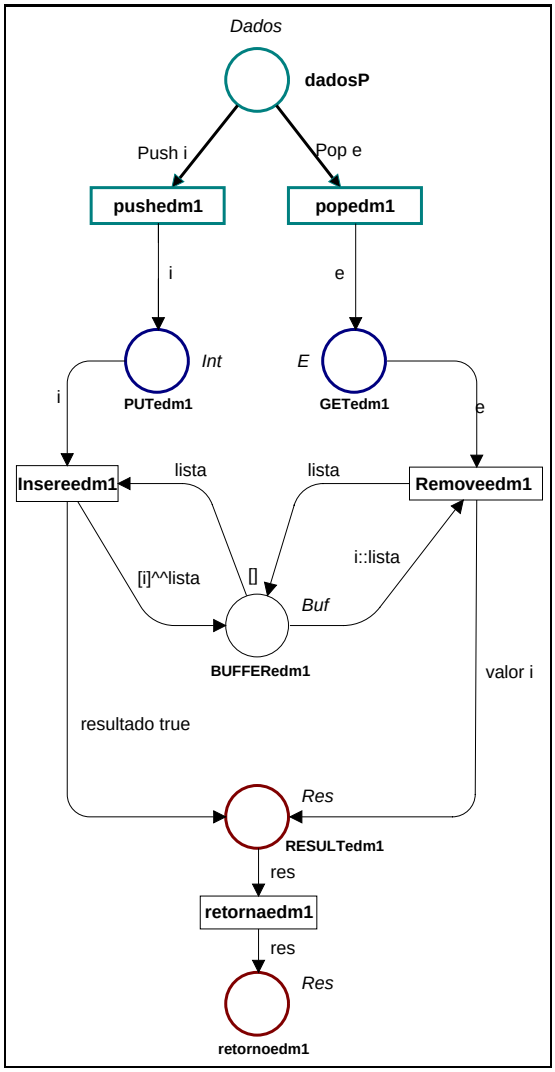

Figura 5.3: Modelo CPN de uma estrutura de dados do tipo pilha.

No modelo da Figura 5.3 tem-se a descrição, dada através de uma rede de Petri Colorida, de uma estrutura de dados do tipo pilha. Este modelo deve ser integrado a algum projeto através de transições de substituição, e tem como portas de entrada e saída os lugares **dadosP** e **retornoedm1**. Uma ficha colocada no lugar **dadosP** representa uma solicitação para empilhamento ou desempilhamento, dependendo do valor da ficha, de um dado (no caso números inteiros) na pilha.

Se o valor da ficha for um número inteiro, tem-se um requisição de empilhamento

habilitando a transição **Insereedm1**. Quando esta transição ocorrer, o dado será empilhado. A pilha é representada pela ficha em **BUFFERedm1**. Esta ficha é uma estrutura especificada em SML [Ull98] do tipo lista, e sempre que um elemento é adicionado a pilha ele será inserido no início da lista. Após a inserção de um ítem na pilha, é colocada uma ficha em **RESULTedm1** indicando sucesso na operação.

Se, por outro lado, o valor da ficha em **dadosP** for do tipo E tem-se uma requisição de desempilhamento. Nesta caso **Removeedm1** será habilitada, e, quando da sua ocorrência, o valor da cabeça da lista em **BUFFERedm1** será removido. Além disso, o valor removido é devolvido para o ambiente no qual o modelo está integrado.

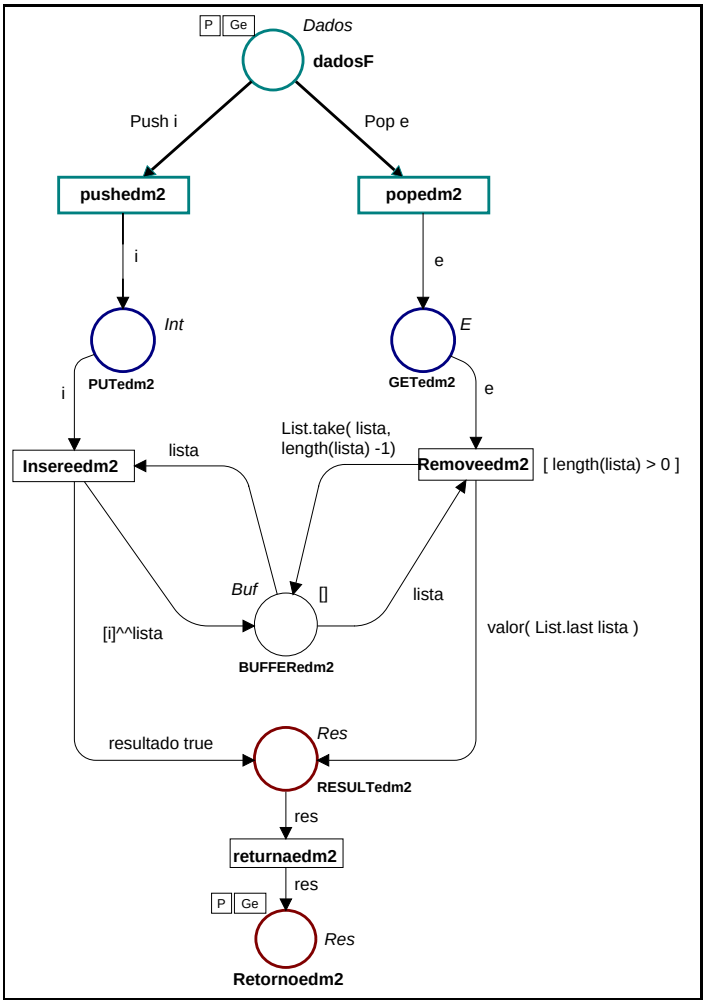

Figura 5.4: Modelo CPN de uma estrutura de dados do tipo fila.

O modelo da Figura 5.4 é bastante parecido com o da Figura 5.3. A principal

diferença, além dos nomes das transições e dos lugares, é que os dados são adicionados no início da lista em **BUFFERedm2** e retirados do final dela (e não do início). Além disso, existe uma guarda na transição **Removeedm2** inibindo sua ocorrência no caso da fila estar vazia.

Deve-se observar que esses modelos de filas e de pilhas não limitam o número máximo de ítens que cada estrutura comporta. Entretanto, empilhadeiras e esteiras, que podem ser modeladas por pilhas e filas, possuem uma capacidade máxima que não deve ser excedida. E é desejável que os modelos que as representam reflitam esta limitação.

Desta forma, é necessário integrar o mecanismo de adaptação de modelos introduzido no Capítulo 4 ao mecanismo de recuperação de modelos de maneira que o modelo recuperado já possua as restrições de comportamento desejadas.

Para que esta integração seja realizada, é preciso efetuar os seguintes passos antes do modelo ser exportado:

1. Descrever as propriedades (restrições de comportamento) desejadas do modelo.
2. Verificar se o modelo pode ser adaptado de maneira a satisfazer tais propriedades;
3. Em caso positivo, adaptar o modelo em questão.

O primeira passo está relacionado com a necessidade do projetista especificar utilizando a linguagem ASK-CTL quais são as restrições que devem ser impostas ao modelo a ser utilizado. No caso do sistema de transporte entre células de produção deseja-se limitar o tamanho máximo das pilhas e filas para que estes reflitam a capacidade físicas das máquinas a serem utilizadas na construção do sistema real.

A descrição das restrições desejadas é feita em duas etapas. Primeiramente deve-se descrever o predicado desejado como uma função que recebe como parâmetro um nó do grafo de ocorrência e retorna um valor booleano. Esta função deve avaliar o nó, determinando se a propriedade desejada é satisfeita ou não. Em seguida escreve-se

a fórmula ASK-CTL expressando o comportamento desejado do sistema em termos destes predicados.

A seguir apresenta-se o predicado ASK-CTL que expressa a necessidade de limitar-se a capacidade da estrutura do tipo pilha em no máximo dois ítens.

```
fun MaximoLimite(n) =
(
let
        val lista_s = ref "";
        val lista_c = ref []
in
        lista_s := (st_Mark.Pilha'BUFFERedm1 1 n);
        lista_c := explode(!lista_s);

        if (conta(fc(!lista_c, #"["), 0) < 3)
        then
                true
        else
                false
end);
```

Observe que para este predicado, **BUFFERedm1** é o lugar do modelo da pilha que contem a lista com os ítens inseridos. Além disso, deve-se notar que este predicado aplica-se apenas ao modelo da pilha. Para a fila é preciso definir um outro predicado:

```
fun MaximoLimite(n) =
(
let
        val lista_s = ref "";
        val lista_c = ref []
in
        lista_s := (st_Mark.Fila'BUFFERedm2 1 n);
        lista_c := explode(!lista_s);

        if (conta(fc(!lista_c, #"["), 0) < 3)
        then
                true
        else
                false
end);
```

A seguir apresenta-se a fórmula ASK-CTL que expressa a propriedade que indica um limite máximo de dois ítens para qualquer uma das duas estruturas, neste caso dois de acordo com os predicados acima.

$$\boxed{POS(NOT(NF(\text{“posso limitar o buffer?”}, MaximoLimite)))}$$

Após descrever às restrições de comportamento desejadas, o projetista deve iniciar a execução do procedimento de adaptação de modelos e aguardar os resultados desta execução.

Como resultado da execução do procedimento de adaptação sobre os modelos e a propriedade acima apresentados obtêm-se os modelos das Figuras 5.5 e 5.6.

5.3 Algumas Considerações

Um outro aspecto relevante no processo de reúso de modelos é como utilizar tais modelos adequadamente. No cenário descrito neste Capítulo considerou-se a existência de ferramentas que suportassem as atividades de armazenamento e recuperação e adaptação de modelos reusáveis. De fato, tal ferramenta já existe e é detalhada em [Lem01]. Entretanto nada discutiu-se sobre a automatização da integração desses modelos.

Como dito anteriormente, a integração é feita através dos mecanismos de lugares de fusão e transições de substituição fornecidos pelo formalismo das Redes de Petri Hierárquicas.

Foge ao escopo deste trabalho detalhar um mecanismo de integração. Entretanto pode-se traçar algumas linhas para fazê-lo. Em primeiro lugar é preciso adicionar todas as declarações de cores de lugares, variáveis e funções utilizadas pelo modelo objeto de reúso ao nó de declaração global do modelo em construção. Depois, é preciso incluir os sub-diagramas (modelo exportado) ao projeto em andamento através de transições de substituição, assim é preciso configurar automaticamente a qual página a transição deve ser associada e quais os lugares serão portas de entrada

e saída dessa transição. Além disso é preciso dispor de um ambiente de integração para que os modelos recuperados sejam devidamente utilizados.

Um outro aspecto relevante que merece comentários relaciona-se com a natureza composicional dos modelos. Uma vez que os modelos a serem reusados já forma verificados, pode-se confiar em seu correto funcionamento. Assim, quando da análise do modelo em desenvolvimento não é preciso preciso analisar o funcionamento dos modelos que foram integrados. Isso trás uma grande vantagem pois não é necessário gerar o espaço de estados referentes a estes modelos. Desta forma promove-se uma diminuição no tamanho deste espaço de estados, e os mecanismos de análise e verificação a serem utilizados terão que processar menos informação durante suas execuções.

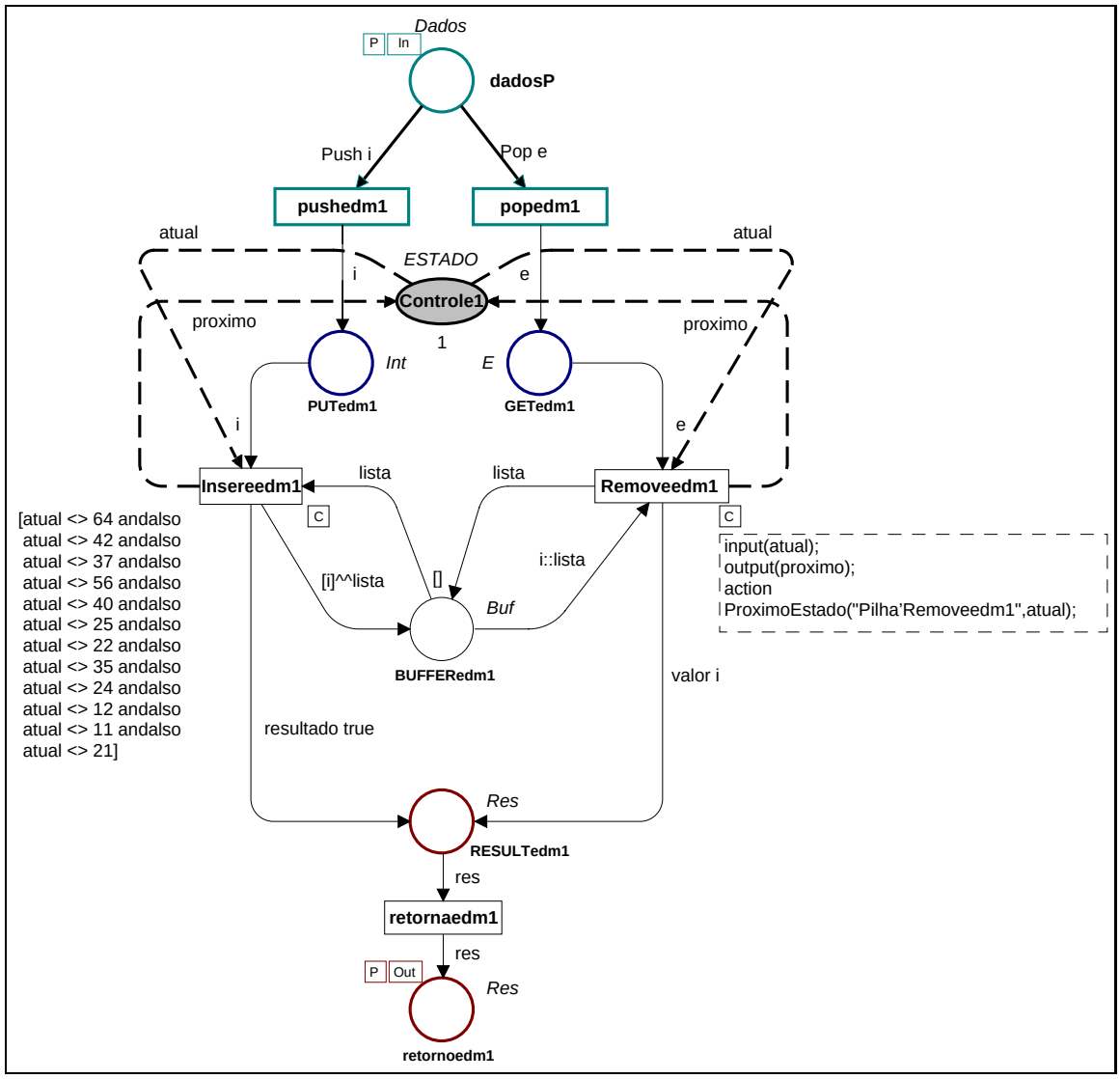

Figura 5.5: Modelo CPN de uma estrutura de dados do tipo pilha com restrições de controle.

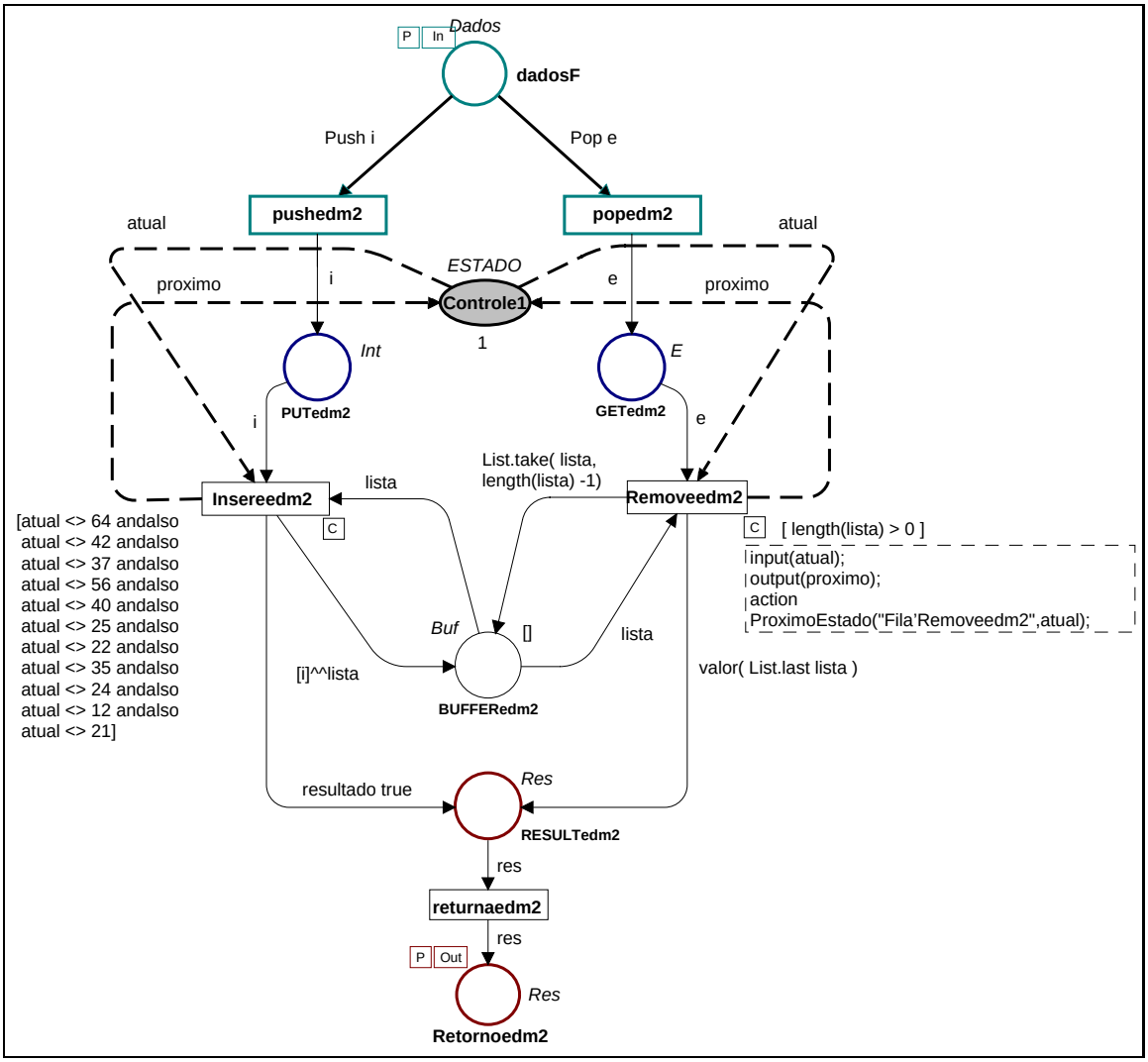

Figura 5.6: Modelo CPN de uma estrutura de dados do tipo fila com restrições de controle.

Capítulo 6

Conclusões e Trabalhos Futuros

Nos capítulos anteriores introduziu-se uma abordagem para a adaptação de modelos em redes de Petri Coloridas baseada na Teoria do Controle Supervisório (TCS) e na técnica de Verificação Automática de Modelos.

Esta abordagem permite que, dado um modelo descrito por uma rede de Petri Colorida e um conjunto de restrições de comportamento desejadas, seja construído um novo modelo que satisfaça às restrições definidas.

Além disso, esta abordagem encaixa-se em um contexto de reúso em que ainda estão previstas atividades de armazenamento e recuperação de modelos.

Para o desenvolvimento deste trabalho, em um primeiro momento, realizou-se uma pesquisa objetivando identificar conceitos e mecanismos utilizados no contexto de reúso de código que pudessem ser aplicados para o reúso de modelos. Para tanto, estudou-se as categorias de reúso e suas abordagens.

Além disso, tendo em vista o foco principal deste trabalho de dissertação, foram pesquisados arcabouços formais para dar suporte à definição, especificação e realização de um mecanismo automático para adaptação de modelos. Dentre os formalismos estudados, optou-se por utilizar os conceitos e técnicas da TCS e da Verificação Automática de Modelos para definir a abordagem de adaptação relatada neste trabalho.

Uma vez estabelecida a abordagem a ser seguida para a adaptação de modelos,

definiu-se qual o ambiente operacional que deveria dar suporte a esta abordagem. Dentre os diversos ambientes disponíveis optou-se pela utilização do Design/CPN e da biblioteca ASK-CTL [JCHH91; CM96]. O procedimento descrito no Capítulo 4 foi implementado em SML'97 [Ull98] tendo como ambiente operacional o próprio Design/CPN.

Num primeiro momento a solução adotada foi implementada considerando a existência de um modelo a ser adaptado isoladamente, como relatado no Capítulo 4. No entanto, a adaptação é apenas um aspecto dentro do contexto de reúso, outros aspectos importantes são armazenamento e recuperação. Tais aspectos foram considerados em [Lem01] para definição e implementação de um método para o armazenamento e uma técnica para recuperação de modelos descritos em redes de Petri Coloridas.

Desta forma, integrou-se a implementação para o caso isolado à solução para armazenamento e recuperação, como relatado no Capítulo 5. Para ambas as situações exemplificou-se a utilização da abordagem definida.

Um aspecto relevante no processo de reúso de modelos que não foi considerado no contexto deste trabalho é como utilizar tais modelos adequadamente. Uma possível solução para isto é a definição e implementação de um mecanismo automático para a integração de modelos de modo a minimizar a intervenção humana neste processo, evitando, assim, que erros possam ser introduzidos nos modelos recuperados e adaptados.

Além disso, a construção de fórmulas em lógica temporal mostrou-se não ser uma operação trivial. Algumas soluções podem ser implementadas para amenizar este problema. Entre elas destaca-se a possibilidade de um sistema de padrões de propriedades para descrevê-las [DAC98; Pen98]. De qualquer forma, é preciso exercitar o uso do ASK-CTL para descrever os padrões existentes em termos de propriedades do espaço de estados das redes de Petri Coloridas. Além disso, um assistente de edição de fórmulas ASK-CTL pode ser construído para auxiliar o projetista na tarefa de descrever propriedades.

Mais que um assistente de edição de fórmulas, é possível definir uma ferramenta gráfica para guiar o projetista por toda a atividade de reúso, desde o armazenamento de modelos até a integração destes, passando pela recuperação e pela adaptação. De posse de uma ferramenta desta natureza é possível automatizar boa parte das atividades de reúso, diminuindo a sobrecarga cognitiva para o projetista. Além disso, é possível, através de mecanismos de comunicação em redes de computadores, efetuar a busca por modelos reusáveis em um repositório de modelos distribuído.

Ainda com relação ao ASK-CTL, é preciso dizer que o verificador de modelos desta linguagem apenas informa ao término de sua execução se as propriedades especificadas são satisfeitas ou não. Ainda não foi implementado o mecanismo para gerar um contra-exemplo quando a propriedade é falsa. Alterar esta biblioteca para ter acesso aos contra-exemplos é interessante para o contexto de reúso de modelos no qual se insere este trabalho.

Além disso, também propõe-se a investigação de mecanismos que possibilitem modificar a biblioteca ASK-CTL para que a adaptação dos modelos em questão seja realizada durante o processo de verificação automática.

Um outra aspecto relevante a ser considerado, é a utilização da abordagem definida neste trabalho no contexto de reúso de código.

Bibliografia

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991. Winner of the Sidney Michaelson Best Paper Award.

[BCL+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.

[BP89a] T. Biggerstaff and A. Perlis. *Software Reusability: Applications and Experience*, volume II of *Frontier Series*. ACM Press, New York, 1989.

[BP89b] T. Biggerstaff and A. Perlis. *Software Reusability: Concepts and Models*, volume I of *Frontier Series*. ACM Press, New York, 1989.

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

[CCM97] Allan Cheng, Søren Christensen, and Kjeld Høyer Mortensen. Model checking coloured petri nets exploiting strongly connected components. Technical report, Computer Science Department, Aarhus University, Aarhus C, Denmark, March 1997.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights*, volume 131 of *Lectures Notes in Computer Science*, New York, May 1981. Springer-Verlag.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions of Programming Languages and Systems*, 8(2):244–263, 1986.

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

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

[CJ95] Søren Christensen and Kurt Jensen. *The Design/CPN Occurrence Graph Tool - User's Manual version 1.0.* Computer Science Department, Aarhus University, Aarhus, Denmark, 1995.

[CJGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CM96] Søren Christensen and Kjeld Høyer Mortensen. *Design/CPN ASK-CTL Manual*, 1996.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28, December 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.

[dCB00] Tomaz de Carvalho Barros. *Supervisão de Sistemas de Produção Baseada em Redes de Petri.* Tese de doutorado, Coordenação de Pós-graduação em Engenharia Elétrica, Universidade Federal da Paraíba, Campina Grande, PB, February 2000.

[dM00] Ana Karla Alves de Medeiros. Mecanismo de interação para um modelo de redes de petri orientado a objetos. Dissertação de mestrado, Coordenação de Pós-graduação em Informática, Universidade Federal da Paraíba, Campina Grande, PB, August 2000.

[dSR99] Zilma Betânia de Sá Ribeiro. Supervisores distribuídos para sistemas flexíveis de manufatura. Dissertação de mestrado, Coordenação de Pós-graduação em Informática, Universidade Federal da Paraíba, Campina Grande, PB, April 1999.

[ES88] E. Allen Emerson and Jai Srinivasan. Branching time temporal logic. In de Bakker J. W, W. P. de Roever, and G. Rozemberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lectures Notes in Computer Science*, pages 123–172. Springer-Verlag, 1988.

[Fre87] Peter Freeman. *Tutorial: Software Reusability.* IEEE Computer Society Press, Washington, D.C., 1987.

[Gen87] H. J. Genrich. Predicate/transition nets. In W. Brauer, W. Reisig, and G. Rozemberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lectures Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[GJM91] Carlo Gezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of*

Software Engineering. Prentice-Hall International Editions, Englewood Cliffs, New Jersey 07632, 1991.

[GP00] Kyller Costa Gorgônio and Angelo Perkusich. Síntese de especificações em redes de petri para componentes de software. In *III Workshop de Métodos Formais*, pages 139 – 144, João Pessoa, PB - Brasil, October 2000.

[Gue97] Dalton Dario Serey Guerrero. Sistemas de redes de petri modulares baseadas em objetos. Dissertação de mestrado, Coordenação de Pós-graduação em Informática, Universidade Federal da Paraíba, Campina Grande, PB, 1997.

[HH79] J. E. Hopcroft and J. D. Hullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[JCHH91] Kurt Jensen, Søren Christensen, P. Huber, and M. Holla. *Design/CPN. A Reference Manual*. Meta Software Corporation, Cambridge Park Drive, USA, 1991.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *EACTS – Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[Jen95] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 2 of *EACTS – Monographs on Theoretical Computer Science*. Springer-Verlag, 1995.

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

[Kre99] Rob Kremer. Cpsc 451: Practical software engineering. http://sern.ucalgary.ca/courses/CPSC/451/W99/, 1999.

[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.

[Lem01] Adriano José Pinheiro Lemos. Reúso de modelos em redes de petri coloridas. Dissertação de mestrado, Coordenação de Pós-graduação em Informática, Universidade Federal da Paraíba, Campina Grande, PB, March 2001.

[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*, pages 223–229, Kyoto, Japan, April 1998.

[MBC+95] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceshinis. *Modeling with Generalized Stochastic Petri Nets*. John Wiley and Sons, 1995.

[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 Co.

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

[Mye79] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[NJ98] Martin Naedele and Jorn W. Janneck. Design patterns in petri net system modeling. In *Proceedings of ICECCS'98*, pages 47–54, October 1998.

[Pen98] John J. Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications*. PhD thesis, University of Cincinnati, April 1998.

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

[RBV94] D. Ribot, B. Blongard, and C. Villermain. Development life-cycle with reuse. In *ACM Symposium on Applied Computing SAC 94*, March 1994.

[RW87a] P. J. G. Ramadge and W. M. Wonham. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987.

[RW87b] P. J. G. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[RW89] P. J. G. Ramadge and W. M. Wonham. The control os discrete event systems. *Proceedings of the IEEE*, 77(1):81–97, 1989.

[SG96] M. Shaw and David Garlan. *Software Architecture. Perspectives of an Emerging Discipline*. Prentice Hall, 1996.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Thi87] P. S. Thiagarajan. Elementary net systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri-Nets: Central Models and their Properties, Proc. of and Advance Course on Petri Nets*, volume 254 of *Lectures Notes in Computer Science*, pages 26–59. Springer-Verlag, 1987.

[Tra88] W. Tracz. *Tutorial: Software Reuse. Emerging Technology*. IEEE Computer Society Press, Los Alamitos, CA, 1988.

[Ull98] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, 2 edition, 1998.

[vdAB96] 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.