CHRISTIANNE REISER

O Ambiente GRAIL para Controle Supervisório de Sistemas a Eventos Discretos: Reestruturação e Implementação de Novos Algoritmos

Florianópolis, setembro de 2005.

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE PÓS-GRADUAÇÃO
EM ENGENHARIA ELÉTRICA

O Ambiente GRAIL para Controle Supervisório de Sistemas a Eventos Discretos: Reestruturação e Implementação de Novos Algoritmos

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

Christianne Reiser

Florianópolis, setembro de 2005.

O Ambiente GRAIL para Controle Supervisório de Sistemas a Eventos Discretos: Reestruturação e Implementação de Novos Algoritmos

Christianne Reiser

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Controle, Automação e Informática Industrial*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.'
Florianópolis, 28 de setembro de 2005.

Prof. José Eduardo Ribeiro Cury, Dr.
Orientador

Prof. Rômulo Silva de Oliveira, Dr.
Co-Orientador

Alexandre Trofino Neto, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

José Eduardo Ribeiro Cury, Dr.
Presidente

Rômulo Silva de Oliveira, Dr.

Antonio Eduardo Carrilho da Cunha, Dr.

Eduardo Camponogara, Dr.

Jean-Marie Farines, Dr.

Agradecimentos

Muitas pessoas foram importantes para a realização deste trabalho.

Inicialmente, gostaria de agradecer o professor José Eduardo Ribeiro Cury pela sua orientação e assistência no decorrer deste trabalho.

Agradeço também o meu co-orientador, o professor Rômulo Silva de Oliveira, e os meus colegas Max Hering de Queiroz e Antonio Eduardo Carrilho da Cunha, os quais me auxiliaram e ajudaram a tornar possível a realização deste trabalho.

Não poderia deixar de agradecer também os meus colegas Patrícia Pena, Tatiana Garcia, Gustavo Bouzon e Julio Antônio Massotti, por estarem sempre presentes quando precisei.

Por último, porém, com a maior importância, quero agradecer a toda a minha família, especialmente meus pais, Osy Reiser e Lilian Hosang, e os meus irmãos, Rafael e Carlos Eduardo Reiser. Eles sempre me ajudaram nos momentos difíceis e me deram o apoio necessário para que eu pudesse alcançar os meus objetivos.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

O Ambiente GRAIL para Controle Supervisório de Sistemas a Eventos Discretos: Reestruturação e Implementação de Novos Algoritmos

Christianne Reiser

setembro/2005

Orientador : José Eduardo Ribeiro Cury
Área de Concentração : Controle, Automação e Informática Industrial
Palavras-chave : Sistemas a Eventos Discretos, Controle Supervisório, Redução de Supervisores, Grail, Sistemas Condição/Evento, Sistemas a Eventos Discretos com Marcação Flexível, Controle Multitarefa
Número de Páginas : ix + 120

O Grail é um ambiente computacional que lida com a computação simbólica de linguagens finitas, expressões regulares e máquinas de estados finitos. Este vem sendo utilizado como uma ferramenta de apoio para o estudo da Teoria de Controle Supervisório. A presente Dissertação de Mestrado introduz uma nova estrutura para o ambiente Grail, tornando-o mais modular. Dentre as contribuições, cita-se a implementação de novas funcionalidades, tal como a verificação de não-conflito entre supervisores modulares e a redução de supervisores. Como a Teoria de Controle Supervisório atua como base para vários ramos de expansão que lidam com problemas diferenciados, a estrutura do Grail possui uma base e seus ramos são definidos por intermédio de módulos. Outra contribuição deste trabalho é o desenvolvimento dos módulos Condição/Evento, Hierárquico e Multitarefa. Estas lidam com Sistemas Condição/Evento, Sistemas a Eventos Discretos com Marcação Flexível e Autômatos com Marcação Colorida, respectivamente.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

The Environment GRAIL for Supervisory Control of Discrete Events Systems: Reorganization and Implementation of New Algorithms

Christianne Reiser

september/2005

Advisor	: José Eduardo Ribeiro Cury
Area of Concentration	: Control, Automation and Industrial Computing
Key words	: Discrete Event System, Supervisory Control, Supervisor Reduction, Grail, Condition/Event System, Multitasking Control, Discrete Event System with Flexible Marking
Number of Pages	: ix + 120

The Grail is a computational environment that deals with the symbolic computation of finite languages, regular expressions and finite state machines. This have been used as a support tool for the study of the Supervisory Control Theory. This Master Thesis introduces a new structure for the tool, making it more modular. Among the contributions, there is the implementation of new functionalities for the tool, such as the verification of non-conflict between modular supervisors and the reduction of supervisors. As the Supervisory Control Theory has many extensions that deal with different kinds of problems, the proposed structure of Grail has a base and branches, these defined by toolboxes. Another contribution of this work is the development of the Condition/Event, Hierarchic and Multitasking toolboxes, which deal with Condition/Event Systems, Discrete Event System with Flexible Marking and Colored Marking Automata, respectively.

Sumário

1 **Introdução** 1

1.1 Teoria de Controle Supervisório 2

1.1.1 Sistemas Condição/Evento 3

1.1.2 Abordagem Hierárquica 3

1.1.3 Controle Multitarefa 3

1.2 Computação Aplicada à TCS 4

1.3 Objetivos 5

1.4 Estrutura da Dissertação 5

2 **A Teoria de Controle Supervisório** 6

2.1 Linguagens como Modelos para SEDs 6

2.1.1 Operações sobre Linguagens 6

2.1.2 Representação de SEDs por Linguagens 8

2.1.3 Expressões Regulares 8

2.2 Autômatos como Modelos para SEDs 8

2.2.1 Operações sobre Autômatos 9

2.2.2 Representação de SEDs por Autômatos 11

2.3 Controle Supervisório de SEDs 11

2.3.1 Abordagem Monolítica 12

2.3.2 Abordagem Modular 13

2.3.3 Redução de Supervisores 15

2.4 Exemplos 16

2.4.1 O Gato e o Rato num Labirinto 16
2.4.2 Célula de Manufatura 17
2.5 Conclusões 21

3 O Grail e sua Nova Estrutura **22**
3.1 O Grail Versão 2.5 23
3.1.1 Conceitos Básicos 23
3.1.2 Organização dos Arquivos 25
3.1.3 Implementação 25
3.2 Evolução do Ambiente Grail 33
3.3 A Nova Estrutura do Grail 34
3.3.1 Principais Modificações 34
3.3.2 Módulos 35
3.3.3 Extensão do Grail para Controle Supervisório 36
3.4 Conclusões 39

4 O Grail para Controle Supervisório **41**
4.1 Filtros 41
4.1.1 Filtros do Grail Versão 2.5 41
4.1.2 Filtros Referentes ao Controle Supervisório 47
4.1.3 Novos Filtros 50
4.2 Exemplo - Mesa Giratória 55
4.2.1 Síntese do Supervisor Monolítico 57
4.2.2 Síntese dos Supervisores Modulares Locais 59
4.3 Conclusões 61

5 Controle Multitarefa **62**
5.1 Comportamento Colorido 62
5.2 Autômato de Marcação Colorida 63
5.2.1 Linguagens Associadas a um AMC 64

5.2.2 Propriedades de AMCs 64
5.2.3 Operações sobre AMCs 65
5.2.4 Bloqueio 66
5.3 Controle Supervisório Multitarefa 67
5.3.1 Supervisor Incolor 68
5.3.2 Supervisor Pintor 69
5.3.3 Existência de Supervisores 70
5.3.4 Abordagem Monolítica 71
5.3.5 Abordagem Modular 71
5.4 Grail - Módulo Multitarefa 76
5.4.1 Classes 76
5.4.2 Filtros 78
5.5 Exemplo - Labirinto do Gato e do Rato 90
5.6 Conclusões 92

6 Controle Supervisório de SEDs com Marcação Flexível 93

6.1 Noções Preliminares 93
6.2 Controle Supervisório de SEDMFs 95
6.2.1 Exemplo - Um Gato e um Rato num Labirinto 96
6.3 Grail - Módulo Hierárquico 98
6.3.1 Classes 99
6.3.2 Filtros 101
6.4 Conclusões 102

7 Sistemas Condição/Evento 103

7.1 Modelagem de SCEs 104
7.2 SCEs como Modelos para SEDs 105
7.3 Controle Supervisório de SCE 107
7.3.1 Abordagem Monolítica 107
7.3.2 Abordagem Modular 111

7.4 Grail - Módulo Condição/Evento 113
7.4.1 Classes 113
7.4.2 Filtros 114
7.5 Conclusões 115

8 Conclusões e Perspectivas 116

Lista de Símbolos e Abreviaturas

SED	Sistema a Eventos Discretos
SCE	Sistemas Condição/Evento
SEDMF	Sistema a Eventos Discretos com Marcação Flexível
SEDMT	Sistema a Eventos Discretos Multitarefa
L	Linguagem
L	Linguagem Gerada
L_m	Linguagem Marcada
K	Linguagem-Alvo
FL	Linguagens Finitas
RE	Expressão Regular
A	Autômato
AMC	Autômato com Marcação Colorida
P	Autômato que representa a Planta
E	Autômato que representa a Especificação
S	Autômato que representa o Supervisor
S_{red}	Autômato que representa o Supervisor Reduzido
Q	Conjunto de Estados
Σ	Alfabeto - Conjunto de Eventos
Σ_c	Conjunto de Eventos Controláveis
Σ_u	Conjunto de Eventos Não-Controláveis
Q_m	Conjunto de Estados Marcados
FM	Máquina de Estados Finitos
CFM	Máquina de Estados Finitos Colorida
FFM	Máquina de Estados Finitos Flexível
Q_i	Conjunto de Estados Iniciais da FM
Θ	Conjunto de Instruções da FM
θ	Instrução da FM
θ_{source}	Estado Fonte da Instrução θ
θ_{event}	Etiqueta de Transição da Instrução θ
θ_{sink}	Estado Destino da Instrução θ
Q_f	Conjunto de Estados Finais da FM

Capítulo 1

Introdução

A atual tendência de globalização da economia em países industrializados tem acirrado a concorrência entre as empresas, provocando uma busca incessante por maior qualidade e menor custo dos produtos e serviços. Com isso, a eficiência e a flexibilidade dos meios produtivos e gerenciais têm sido fatores decisivos ao sucesso das empresas. A busca por competitividade, aliada à crescente escassez dos recursos naturais e a valorização da mão-de-obra, tem justificado grandes esforços na otimização e automação flexível dos processos [de Queiroz, 2004].

Esse contexto tem favorecido o surgimento de sistemas dinâmicos cada vez mais complexos e que envolvem diversas aplicações, tais como redes de computadores, sistemas automatizados de manufatura, robótica, controle de tráfego, automação predial, entre outras. Entre esses sistemas, destaca-se a classe de Sistemas a Eventos Discretos (SEDs), que são caracterizados por uma dinâmica dirigida pela ocorrência de eventos. São exemplos de eventos discretos a ativação de um sensor e o início de operação de uma máquina. A ocorrência desses eventos não depende diretamente da passagem do tempo, mas sim de uma mudança discreta no estado do sistema.

A natureza discreta dos SEDs faz com que os modelos matemáticos convencionais, baseados em equações diferenciais, não sejam adequados para tratá-los. Por outro lado, a sua importância faz com que seja altamente desejável encontrar soluções para problemas relacionados ao seu controle. Em razão disso, existe uma intensa atividade de pesquisa voltada à busca de modelos matemáticos adequados à sua representação, sem que se tenha conseguido até agora encontrar um modelo que seja matematicamente tão conciso e computacionalmente tão adequado como o são as equações diferenciais para os sistemas dinâmicos de variáveis contínuas. Dentre os modelos existentes, destaca-se o proposto por Ramadge e Wonham, baseado na Teoria de Linguagens e Autômatos e aqui denominada Teoria de Controle Supervisório (TCS).

1.1 Teoria de Controle Supervisório

A TCS faz uma distinção clara entre o sistema a ser controlado, denominado planta, e a entidade que o controla, que recebe o nome de supervisor. A planta é um modelo que reflete o comportamento fisicamente possível do sistema, isto é, todas as ações que este é capaz de executar na ausência de qualquer ação de controle. Em geral, este comportamento inclui a capacidade de realizar determinadas atividades que produzam um resultado útil, sem contudo se limitar a este comportamento desejado. Por exemplo, dois robôs trabalhando em uma célula de manufatura podem ter acesso a um depósito de uso comum, o que pode ser útil para passar peças um ao outro. No entanto, cria-se com isso a possibilidade física de ocorrer um choque entre ambos, o que é indesejável. O papel do supervisor é, então, o de exercer uma ação de controle restritiva sobre a planta, de modo a confinar seu comportamento àquele que corresponde a uma dada especificação [Cury, 2001].

Uma vantagem desta abordagem é a de permitir a síntese de supervisores, sendo estes obtidos de forma a restringir o comportamento da planta apenas o necessário para evitar que esta realize ações proibidas. Desta forma, pode-se verificar se uma dada especificação de comportamento pode ou não ser cumprida e, caso não possa, identifica-se a parte dessa especificação que pode ser implementada de forma minimamente restritiva.

Muito embora a complexidade dos algoritmos de síntese de supervisores referentes à TCS seja polinomial em relação ao número de estados dos sistemas que representam a planta e as especificações, este número varia exponencialmente com o número de subsistemas presentes nos mesmos. Este fator inviabiliza a aplicação dos algoritmos originais de síntese para sistemas de grande porte, tal como um sistema de manufatura real. Estudos vêm sendo realizados com o objetivo de solucionar ou diminuir este problema. Dentre as possíveis soluções encontradas, pode-se citar:

- **Abordagem Modular:** visa reduzir a complexidade computacional da síntese dos supervisores por intermédio da construção de um supervisor para cada restrição, de forma que, atuando em conjunto, estes supervisores satisfaçam a especificação global.
- **Abordagem Hierárquica:** visa reduzir a complexidade computacional por intermédio da decomposição do problema de controle original em problemas menores, hierarquicamente relacionados (Seção 1.1.2).

Além do problema computacional, a explosão combinatória de estados torna um supervisor calculado a partir da TCS ilegível. Como solução deste, cita-se a Redução

de Supervisores. Um supervisor minimamente restritivo incorpora informações redundantes, visto que ele contém restrições já realizadas pela planta. Isto significa que é possível reduzir o número de estados deste supervisor sem afetar sua ação de controle, tornando-o mais legível.

Na literatura, outras extensões da TCS vem sendo desenvolvidas. Extensões cujo objetivo é o refinamento da TCS para o tratamento de problemas específicos. Dentre estas, citam-se os Sistemas Condição/Evento e o Controle Multitarefa, introduzidos nas Seções 1.1.1 e 1.1.3.

1.1.1 Sistemas Condição/Evento

Os Sistemas Condição/Evento podem ser vistos como uma classe de SEDs que permitem a modelagem do sistema como a interconexão de subsistemas com sinais de entrada e saída discretos. Nesta extensão, pode-se modelar o sistema de modo que eventos sejam associados a saídas da planta e comandos gerados pelo supervisor sejam associados a entradas da planta, denominadas condições. Em muitos casos, esta metodologia de modelagem mostra-se mais intuitiva e natural que aquela baseada no modelo de Ramadge e Wonham. Além disso, ela possibilita que a modelagem de SEDs baseie-se em diagramas de blocos e em fluxos de sinais, tal como normalmente ocorre na teoria de sistemas [Leal, 2005].

1.1.2 Abordagem Hierárquica

A Abordagem Hierárquica explora a decomposição vertical da arquitetura do sistema. Para que uma estrutura hierárquica de controle seja eficaz, é necessário que haja consistência hierárquica entre os níveis da mesma. Essencialmente, dois níveis são hierarquicamente consistentes quando os comandos aplicados a um dado nível da hierarquia produzem os resultados esperados no nível inferior.

O trabalho de [da Cunha, 2003] teve como objetivo o desenvolvimento de uma nova proposta de hierarquia de dois níveis com consistência hierárquica. Na abordagem de [da Cunha, 2003], o nível inferior é modelado pelo modelo de Ramadge e Wonham e o superior é modelado por uma nova classe de SEDs, os Sistemas a Eventos Discretos com Marcação Flexível (SEDMF) [Cury et al., 2004].

1.1.3 Controle Multitarefa

Em muitos problemas reais, diversos tipos diferentes de tarefas são executados. Nestas situações, a modelagem da planta por um único autômato pode ser proble-

mática, já que a realização de qualquer tipo de tarefa seria identificada pela mesma marcação. O Controle Multitarefa é uma abordagem que trata múltiplos tipos de tarefas no controle supervisório de SEDs.

Quando um SED inclui múltiplos tipos de tarefas, é chamado de Sistemas a Eventos Discretos Multitarefa (SEDMT). Para a modelagem de SEDMTs, introduz-se um novo tipo de autômato, o Autômato com Marcação Colorida, que permite a síntese de supervisores mais refinados em problemas de controle nos quais a distinção de tipos de tarefas é necessária [de Queiroz et al., 2005].

1.2 Computação Aplicada à TCS

A Teoria de Controle Supervisório vêm sendo amplamente estudada e conseqüentemente expandida, tornando-se uma teoria promissora para a resolução de problemas que envolvem Sistemas a Eventos Discretos. A computação se encaixa aqui como uma ferramenta de auxílio, cujo objetivo é facilitar a investigação prática e teórica da TCS e suas extensões.

Neste contexto, estão sendo desenvolvidas diversas ferramentas para a síntese de supervisores para SEDs. Suas implementações ocorrem, em sua maioria, em ambientes acadêmicos e seus objetivos são educacionais. Citam-se aqui as seguintes ferramentas:

- **TCT** [Wonham, 2005], desenvolvida pela Universidade de Toronto;
- **UMDES Software Lybrary** [UMDES, 2005], desenvolvida na Universidade de Michigan;
- **VALID**, desenvolvida na Siemens;
- **CONDES** [Torrico, 1999], desenvolvida na Universidade Federal de Santa Catarina;
- **GRAIL** [Raymond and Wood, 2002], desenvolvida na Universidade de Ontario.

Dentre estas ferramentas, destaca-se o Grail. Este é um ambiente de computação simbólica que envolve linguagens finitas, expressões regulares e máquinas de estados finitos [Raymond and Wood, 1996a]. Foi desenvolvido sob o paradigma da orientação a objeto e implementado em C++. Caracteriza-se por sua modularidade, por sua facilidade de expansão e por se tratar de um ambiente de código aberto.

O Grail versão 2.5 foi sendo expandido, dentro do Departamento de Automação e Sistemas (DAS) da Universidade Federal de Santa Catarina, com o objetivo de focar

sua utilização na Teoria de Controle Supervisório. Esta expansão ocorreu de forma descontrolada e o resultado disso foram versões diferentes e incompatíveis de um Grail não documentado.

1.3 Objetivos

O presente trabalho teve como objetivo a reorganização e a definição de uma estrutura mais adequada à expansão do Grail para funções relacionadas ao controle supervisório de SEDs, sendo o resultado disso o denominado Grail para Controle Supervisório. Dentro dos aspectos da reestruturação do Grail, cita-se a criação de módulos, cuja função é o tratamento das várias extensões da TCS da forma mais modular possível.

Citam-se também como objetivos deste trabalho a implementação de novas funcionalidades para o Grail para Controle Supervisório e a criação dos módulos Condição/Evento, Hierarquico e Multitarefa. Estes lidam com Sistemas Condição/Evento, SEDs com Marcação Flexível e Controle Multitarefa, respectivamente.

1.4 Estrutura da Dissertação

Este documento está organizado da seguinte forma. A Teoria de Controle Supervisório, iniciada por Ramadge e Wonham, é descrita no Capítulo 2. O Capítulo 3 explora tanto o Grail versão 2.5 como a reeestruturação do mesmo. Os filtros do Grail que referem-se à teoria base são apresentados no Capítulo 4. Algumas extensões da TCS, tais como os Sistemas Condição/Evento, os SEDs com Marcação Flexível e o Controle Multitarefa são apresentadas nos Capítulos 7, 6 e 5, respectivamente. Estes capítulos apresentam, além da teoria que cerca as extensões em questão, os módulos referentes as mesmas. São elas: o módulo Condição/Evento, o Hierarquico e o Multitarefa. Por fim, o Capítulo 8 apresenta as conclusões e perspectivas desta dissertação.

Capítulo 2

A Teoria de Controle Supervisório

Neste capítulo, os principais conceitos da Teoria de Controle Supervisório, iniciada por Ramadge e Wonham (1987), são apresentados. Esta teoria aborda o controle de Sistemas a Eventos Discretos (SEDs), os quais podem ser modelados por linguagens. Estas, por sua vez, podem ser representadas por autômatos.

As Seções 2.1 e 2.2 apresentam os elementos básicos da Teoria de Linguagens e de Autômatos de Estados Finitos, respectivamente. A Seção 2.3 trata especificamente do Controle Supervisório de SEDs e algumas extensões e a Seção 2.4 mostra exemplos que envolvem a modelagem de SEDs através de autômatos, indicando a forma de resolução do problema de controle associado a eles. Este capítulo é baseado em [Cury, 2001].

2.1 Linguagens como Modelos para SEDs

Uma linguagem L definida sobre um alfabeto Σ é um conjunto de cadeias formadas pela concatenação de símbolos pertencentes a Σ. Por exemplo, $L_1 = \{\alpha, \beta, \gamma\}$ e $L_2 = \{\alpha, \alpha\beta\beta\}$ são linguagens sobre o alfabeto $\Sigma = \{\alpha, \beta, \gamma\}$.

O conjunto de todas as possíveis cadeias finitas compostas por elementos de Σ é denotado por Σ^* e este inclui a cadeia vazia ϵ. Uma linguagem sobre Σ é, portanto, um subconjunto de Σ^*.

2.1.1 Operações sobre Linguagens

Algumas operações podem ser executadas sobre linguagens. Além das operações usuais sobre conjuntos, como a união e a intersecção, citam-se:

1. `Concatenação:` Dadas duas linguagens L_1,$L_2 \subseteq \Sigma^*$, a concatenação de L_1 e L_2, representada por L_1L_2, é definida por

$$L_1L_2 = \{s \in \Sigma^* : s = s_1s_2, s_1 \in L_1, s_2 \in L_2\} \tag{2.1}$$

2. `Prefixo-Fechamento:` Seja uma linguagem $L \in \Sigma^*$, então, o prefixo-fechamento de L, denotado por $\overline{L}$, é definido por

$$\overline{L} = \{s \in \Sigma^* : \exists t \in \Sigma^*, st \in L\} \tag{2.2}$$

Em palavras, $\overline{L}$ consiste em todas as cadeias de Σ^* que são prefixos de L. L é dita prefixo-fechada se $L = \overline{L}$, ou seja, se qualquer prefixo de qualquer cadeia de L for também uma cadeia de L.

3. `Fechamento-Kleene:` Seja uma linguagem $L \subseteq \Sigma^*$, então o fechamento Kleene de L, dado por L^*, é definido por

$$L^* = \{\epsilon\} \cup L \cup LL \cup LLL \cup ... \tag{2.3}$$

Uma cadeia de L^* é formada pela concatenação de um número finito de cadeias de L, incluindo a cadeia vazia ϵ.

4. `Projeção Natural:` Sejam Σ e Σ_i alfabetos, tal que $\Sigma_i \subset \Sigma$. $P_i : \Sigma^* \rightarrow \Sigma_i^*$, a projeção natural de Σ^* para Σ_i^*, é definida recursivamente por:

$$\begin{aligned} P_i(\epsilon) &= \epsilon \\ P_i(\sigma) &= \begin{cases} \epsilon, caso\ \sigma \notin \Sigma_i \\ \sigma, caso\ \sigma \in \Sigma_i \end{cases} \\ P_i(s\sigma) &= P_i(s)P_i(\sigma)\ s \in \Sigma^*, \sigma \in \Sigma \end{aligned} \tag{2.4}$$

O conceito de projeção natural pode ser estendido para linguagens como $P_i(L) = \{s_i \in \Sigma_i^* : s_i = P_i(s) \text{ para algum } s \in L\}$. A projeção inversa é, então, definida como $P_i^{-1}(L_i) = \{s \in \Sigma^* : P_i(s) \in L_i\}$.

5. **Composição Síncrona:** Sejam $L_i \subseteq \Sigma_i^*$, $i = 1, ..., n$. Seja $\Sigma = \cup_{i=1}^n \Sigma_i$ e $P_i : \Sigma^* \rightarrow \Sigma_i^*$. Define-se a composição síncrona de linguagens $\|_{i=1}^n L_i \subseteq \Sigma^*$ como:

$$\|_{i=1}^n L_i = \cap_{i=1}^n P_i^{-1}(L_i) = \{s \in \Sigma^* | \wedge_{i=1}^n P_i(s) \in L_i\} \tag{2.5}$$

2.1.2 Representação de SEDs por Linguagens

Se considerarmos o alfabeto Σ como correspondendo ao conjunto de eventos que afeta um SED, então o comportamento deste sistema pode ser descrito na forma de um par de linguagens (L, L_m), sendo que:

- $L \subseteq \Sigma^*$ representa o comportamento gerado do sistema, ou seja, possui todas as palavras fisicamente possíveis de ocorrerem no sistema. Esta linguagem é prefixo-fechada.
- $L_m \subseteq L$ descreve o comportamento marcado do sistema. Trata-se de um subconjunto de L que possui apenas as cadeias que correspondem a tarefas completas do sistema.

2.1.3 Expressões Regulares

Uma linguagem pode ser vista como uma maneira formal de descrever o comportamento de um SED. Porém, a descrição de uma linguagem feita pela enumeração das cadeias que a definem é uma tarefa pouco prática. É necessário, portanto, utilizar estruturas compactas que possam representar estas linguagens. Dentre estas estruturas pode-se citar as Expressões Regulares (RE).

Para um alfabeto Σ dado, define-se recursivamente uma RE da seguinte forma:

1. (a) $\emptyset$ é uma expressão regular que representa a linguagem vazia,
   (b) ϵ é uma expressão regular denotando a linguagem $\{\epsilon\}$,
   (c) σ é uma expressão regular denotando $\{\sigma\}$ $\forall \sigma \in \Sigma$;
2. Se r e s são expressões regulares, então rs, r^*, s^*, $(r+s)$[1] são expressões regulares;
3. Toda expressão regular é obtida pela aplicação das regras 1 e 2 um número finito de vezes.

2.2 Autômatos como Modelos para SEDs

Um autômato é uma quíntupla $A = (Q, \Sigma, \delta, q_0, Q_m)$, onde:

- Q representa um conjunto não-vazio e finito de estados;

[1] r união com s.

- Σ é o conjunto de símbolos que definem o alfabeto;
- $\delta : Q \times \Sigma \rightarrow Q$ representa a função de transição definida em alguns ou todos os estados para os símbolos de Σ, não necessariamente todos;
- $q_0 \in Q$ é o estado inicial do autômato;
- $Q_m \subseteq Q$ é o conjunto de estados finais ou marcados.

A função de transição δ pode ser naturalmente estendida para palavras como a função $\delta : Q \times \Sigma^* \rightarrow Q$ tal que, para $q \in Q$, $s \in \Sigma^*$ e $\sigma \in \Sigma$, $\delta(q, \epsilon) = q$ e $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$ sempre que $q' = \delta(q, s)$ e $\delta(q', \sigma)$ estiverem ambas definidas. Além disso, o conjunto de símbolos habilitados em todo estado $q \in Q$ é denotado por $\Sigma(q)$, ou seja, $\Sigma(q) = \{\sigma \in \Sigma : \delta(q, \sigma)!$ [2]$\}$.

A representação de um autômato pode ser feita através de grafos dirigidos, onde os nós representam estados e os arcos etiquetados representam as transições entre os estados. O estado inicial é identificado através de uma seta apontando para ele e os estados finais, ou marcados, são representados por círculos duplos.

O autômato $A = (Q, \Sigma, \delta, q_0, Q_m)$ reconhece duas linguagens, a linguagem gerada $L(A) = \{s \in \Sigma^* : \delta(q_0, s)!\}$ e a linguagem marcada $L_m(A) = \{s \in L(A) : \delta(q_0, s) \in Q_m\}$.

2.2.1 Operações sobre Autômatos

Um autômato A é dito ser acessível se todos os seus estados forem acessíveis. Um estado $q \in Q$ é acessível se $\exists s \in \Sigma^*$ tal que $\delta(q_0, s) = q$, ou seja, se existir pelo menos uma cadeia que, partindo do estado inicial, alcance o estado q.

Um estado $q \in Q$ é coacessível se $\exists s \in \Sigma^*$ tal que $\delta(q, s) \in Q_m$, isto é, se houver uma cadeia aceita por A que, partindo de q, alcance um estado marcado. Diz-se que um autômato é coacessível se todos os seus estados forem coacessíveis e que é *trim* se todos os seus estados forem acessíveis e coacessíveis.

Um autômato não-determinista é aquele que possui um ou mais estados a partir dos quais transições etiquetadas com um mesmo evento alcançam mais do que um estado. Um exemplo é mostrado na Figura 2.1. Pode-se observar que o estado `0` possui duas transições etiquetadas com o evento `a`, sendo que uma retorna ao estado `0` e a outra leva ao estado `1`, o que caracteriza o não-determinismo.

[2] $\delta(q_0, s)$ está definida.

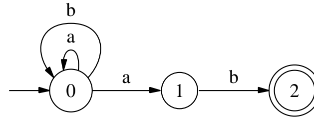


Figura 2.1: Exemplo de um Autômato Não-determinista

Todo autômato não-determinista corresponde a um autômato determinista equivalente, ou seja, que reconhece a mesma linguagem gerada e marcada. O método de construção de um equivalente determinista é apresentado em [Cury, 2001].

Não existe um modelo único para a representação de um dado par de linguagens (L, L_m). No entanto, é desejável, por razões computacionais ou mesmo pela legibilidade, que se trabalhe com autômatos que tenham o menor número possível de estados. Um algoritmo de minimização que agregue estados considerados equivalentes é, portanto, interessante. Dois estados são considerados equivalentes se, dado $A = (Q, \Sigma, \delta, q_0, Q_m)$, temos que:

1. Se $q_1 \in Q_m$, então $q_2 \in Q_m$ (e vice-versa);
2. $\delta(q_1, \sigma) = \delta(q_2, \sigma)$ para todo $\sigma \in \Sigma$.

Um algoritmo para a minimização de autômatos, extensão do algoritmo de Hopcroft, pode ser encontrado em [Cury, 2001], outro é apresentado no Capítulo 4. O autômato mínimo é único, a menos de possível isomorfismo. Dois autômatos são ditos isomorfos se forem iguais a menos da numeração dos estados.

Sejam autômatos A_i, $i = 1, ..., n$. A composição síncrona $A = \|_{i=1}^{n} A_i$ é obtida fazendo-se a evolução em paralelo dos n autômatos A_i, na qual um evento comum a múltiplos autômatos só é executado se todos os autômatos que contiverem este evento o executarem simultaneamente. As linguagens resultantes da composição síncrona são caracterizadas por:

$$L(A) = \|_{i=1}^{n} L(A_i)$$
$$L_m(A) = \|_{i=1}^{n} L_m(A_i)$$

Formalmente, a composição síncrona de dois autômatos $A_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, Q_{m1})$ e $A_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, Q_{m2})$ é definido como o autômato:

$$A_1 \| A_2 = Ac(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{01}, q_{02}), Q_{m1} \times Q_{m2}) \tag{2.6}$$

onde $Ac(A)$ é uma operação que elimina todos os estados não acessíveis de A e:

$$\delta((q_1,q_2),\sigma) = \begin{cases} (\delta_1(q_1,\sigma),\delta_2(q_2,\sigma)), & se\ \sigma \in \Sigma_1 \cap \Sigma_2,\ \delta_1(q_1,\sigma)!,\ \delta_2(q_2,\sigma)! \\ (\delta_1(q_1,\sigma),q_2), & se\ \sigma \in \Sigma_1,\ \sigma \notin \Sigma_2,\ \delta_1(q_1,\sigma)! \\ (q_1,\delta_2(q_2,\sigma)), & se\ \sigma \notin \Sigma_1,\ \sigma \in \Sigma_2,\ \delta_2(q_2,\sigma)! \\ indefinida, & caso\ contrario \end{cases} \quad (2.7)$$

No caso da composição síncrona em que $\Sigma_1 = \Sigma_2$, a linguagem do autômato resultante é a intersecção das linguagens de A_1 e A_2. Ou seja, a intersecção de duas linguagens é a linguagem gerada pelo autômato que representa a ação sincronizada de dois autômatos que operam sobre o mesmo alfabeto.

2.2.2 Representação de SEDs por Autômatos

Se considerarmos o alfabeto Σ como correspondendo ao conjunto de eventos que afeta um SED, então o comportamento deste sistema pode ser descrito de forma que $L(A)$ represente o comportamento gerado e $L_m(A)$ o comportamento marcado do SED em questão. Um autômato *trim* que representa um SED é dito ser não-bloqueante.

2.3 Controle Supervisório de SEDs

Um sistema a ser controlado corresponde, em geral, a um conjunto de subsistemas arranjados segundo uma distribuição espacial dada. Estes subsistemas, vistos isoladamente, têm um comportamento básico original que, quando atuando em conjunto com os demais subsistemas, deve ser restringido de forma a cumprir a função coordenada a ser executada pelo sistema global. A composição dos comportamentos de cada subsistema isolado pode, então, ser identificado como *planta*. Enquanto que o conjunto de restrições de coordenação define uma *especificação* a ser obedecida [Cury, 2001].

Na abordagem proposta por Ramadge e Wonham, a modelagem do comportamento de um SED é feita por intermédio de um autômato, sendo P o autômato que representa a planta e E o que representa a especificação.

De modo a fazer com que os subsistemas atuem de forma coordenada, introduz-se um agente de controle denominado *supervisor*, que direciona a atuação da planta conforme um comportamento desejável por intermédio da desabilitação de eventos possíveis de ocorrer na planta. Entretanto, nem todos os eventos que afetam a planta podem ser desabilitados. O conjunto de eventos da planta Σ é particionado em eventos controláveis Σ_c, que podem ser inibidos de ocorrer no sistema, e eventos não-controláveis Σ_u, cuja natureza não permite a desabilitação.

Um supervisor pode ser representado por um autômato S, definido sobre o mesmo alfabeto que a planta P, cujas mudanças de estado são ditadas pela ocorrência de eventos na planta. A ação de controle de S, definida para cada estado do autômato, é desabilitar em P os eventos que não possam ocorrer em S após uma palavra observada. O funcionamento do sistema controlado S/P é descrito pelo SED resultante da composição síncrona de S e P. De fato, na composição síncrona $S \| P$, somente as transições permitidas na planta P e no supervisor S são permitidas, já que os alfabetos são os mesmos.

Dada uma linguagem $K \subseteq L_m(P)$, não-vazia, representando a linguagem desejada para P, existe um supervisor próprio S que implemente K se e somente se K for $L_m(P)$-fechada e controlável em relação a P. Uma linguagem K é $L_m(P)$-fechada se $K = \overline{K} \cap L_m(P)$, ou seja, se todos os prefixos de K que forem palavras de $L_m(P)$ também forem palavras de K.

Uma linguagem $K \subseteq \Sigma^*$ é controlável em relação a P se $\overline{K}\Sigma_u \cap L(P) \subseteq \overline{K}$. Esta propriedade garante que um supervisor que implemente a linguagem K não vai tentar desabilitar eventos não-controláveis na planta P. O conjunto de linguagens controláveis contidas na linguagem K, denotado por $C(P, K)$, é fechado para união, conseqüentemente, $C(P, K)$ possui um elemento supremo, denominado $SupC(P, K)$, que é a máxima linguagem controlável em relação a P contida em K.

Caso não seja possível construir um supervisor S que implemente K, é possível projetar um supervisor próprio que implemente $SupC(P, K)$. Convém citar que, se K for $L_m(P)$-fechada, então $SupC(P, K)$ também será $L_m(P)$-fechado.

2.3.1 Abordagem Monolítica

A síntese de um supervisor monolítico é feita com base nos seguintes passos.

1. Modelagem do funcionamento do sistema sem coordenação. Este passo inclui a identificação do conjunto de subsistemas que fazem parte da planta e a obtenção do modelo básico P_i de cada subsistema envolvido.

2. Modelagem das restrições de coordenação do sistema. Este consiste na construção de autômatos E_j para cada restrição de coordenação j do sistema a ser controlado.

3. Síntese de uma lógica de controle coacessível e ótima:

   (a) Obtenção da planta através da composição síncrona dos subsistemas P_i;

   (b) Construção da especificação por meio da composição de todas as restrições E_j modeladas;

(c) Composição síncrona da planta P com a especificação E e determinação da componente *trim* desta, obtendo, assim, o autômato que representa a linguagem-alvo K;

(d) Computação do supervisor minimamente restritivo S, cuja linguagem é dada por $SupC(P, K)$. O algoritmo para o cálculo da máxima linguagem controlável por ser encontrado em [Cury, 2001].

A síntese de supervisores compreende, basicamente, os seguintes procedimentos: composição síncrona e cálculo da máxima linguagem controlável. A complexidade computacional destes procedimentos é polinomial ao produto do número de estados da planta P e da especificação E. Entretanto, o número de estados de P e de E cresce exponencialmente em relação ao número de subsistemas e restrições de coordenação, respectivamente. Algumas extensões da abordagem monolítica vêm sendo desenvolvidas na literatura com o objetivo de solucionar ou diminuir este problema. Dentre estas extensões, pode-se citar a Abordagem Modular (Seção 2.3.2) e o Controle Hierárquico [da Cunha, 2003].

Além do problema da complexidade computacional, o supervisor de um sistema de grande porte torna-se ilegível e difícil interpretação por causa do seu grande número de estados. Por disso, verificou-se a possibilidade de redução do número de estados do supervisor sem afetar a ação do controle, conforme apresenta a Seção 2.3.3.

2.3.2 Abordagem Modular

A abordagem modular é uma extensão da monolítica que visa reduzir a complexidade computacional da síntese dos supervisores. Nessa abordagem, ao invés de se projetar um único supervisor que satisfaça todas as restrições, procura-se construir um supervisor para cada restrição, de forma que, atuando em conjunto, os supervisores satisfaçam a especificação global. A ação conjunta de supervisores modulares desabilita um evento controlável sempre que este for desabilitado por pelo menos um deles [Cury, 2001].

A síntese dos supervisores modulares se dá por intermédio da modelagem dos subsistemas e restrições, conforme especificado na Seção 2.3.1, e da síntese dos supervisores modulares conforme os seguintes passos:

1. Obtenção da planta através da composição síncrona dos subsistemas P_i;
2. Composição síncrona da planta P com cada restrição E_j e determinação da componente *trim* desta, obtendo, assim, o autômato que representa a linguagem-alvo correspondente a cada uma das restrições;

3. Computação dos supervisores minimamente restritivos S_j, cuja linguagem é dada por $SupC_j(K_j, P)$.

A síntese por intermédio da abordagem modular propicia uma maior flexibilidade sendo mais fácil de atualizar, modificar e corrigir supervisores que na monolítica. Entretanto, a ação dos diferentes supervisores pode ser conflitante.

Sejam linguagens $L_j \subseteq \Sigma^*$, $j = 1, ..., m$. Diz-se que o conjunto de linguagens L_j é modular, ou não-conflitante, se $\cap_{j=1}^{m} \overline{L_j} = \overline{\cap_{j=1}^{m} L_j}$, ou seja, se um prefixo for aceito por todo o conjunto de linguagens, então todo o conjunto tem que aceitar uma palavra contendo esse prefixo. A condição necessária e suficiente para que o resultado obtido pela abordagem modular seja equivalente à obtida pela monolítica é a modularidade das linguagens marcadas pelas ações dos supervisores S_j.

A abordagem modular é bastante vantajosa, quando a modularidade entre os supervisores é verificada, no sentido de promover maior flexibilidade, maior eficiência computacional e segurança na aplicação do controle. Porém, essa abordagem explora apenas a modularidade das restrições, mas não a da planta. Ambas as modularidades, tanto a das restrições quanto a dos subsistemas da planta, são exploradas na abordagem modular local proposta por [de Queiroz, 2000] e apresentada a seguir.

2.3.2.1 Abordagem Modular Local

Seja a planta P formada por subsistemas $P_i = (Q_i, \Sigma_i, \delta_i, q_{0i}, Q_{mi})$, $i = 1, ..., n$. Sejam, para $j = 1, ..., m$, restrições genéricas E_j definidas, respectivamente, sobre subconjuntos de eventos $\Sigma_j \subseteq \Sigma$. De forma alternativa à abordagem modular clássica, pode-se representar o comportamento desejado do sistema como o conjunto de restrições expressas apenas em termos dos subsistemas que elas restringirem, chamados de plantas locais. Para $j = 1, ..., m$, a planta local $P_{local,j}$ associada a restrição E_j é definida por:

$$P_{local,j} = \|_{i \in I_j} P_i, \; I_j = \{k \in I | \Sigma_k \cap \Sigma_j \neq \emptyset\} \tag{2.8}$$

Assim, a planta local $P_{local,j}$ é composta apenas pelos subsistemas da modelagem original que estão diretamente afetados por E_j.

A síntese de uma lógica de controle ótima por intermédio da abordagem modular local se dá através dos seguintes passos:

1. Obtenção das plantas locais $P_{local,j}$ através da composição síncrona dos subsistemas P_i envolvidos com a restrição E_j;

2. Composição síncrona de cada planta local $P_{local,j}$ com sua respectiva especificação E_j e determinação da componente *trim* desta, obtendo, assim, o autômato que representa a linguagem-alvo $K_{local,j}$ correspondente a cada uma das restrições;

3. Computação dos supervisores locais minimamente restritivos $S_{local,j}$, cuja linguagem é dada por $SupC_{local,j}(K_{local,j}, P_{local,j})$.

A exemplo da abordagem modular clássica, a ação dos diferentes supervisores pode ser conflitante. Sejam linguagens $L_j \subseteq \Sigma^*$, $j = 1, ..., m$. Diz-se que o conjunto de linguagens L_j é localmente modular se $\|_{j=1}^{m} \overline{L_j} = \overline{\|_{j=1}^{m} L_j}$. Em outras palavras, sendo as linguagens L_j marcadas por autômatos $S_{local,j}$, a modularidade local é verificada se e somente se $S = \|_{j=1}^{m} S_{local,j}$ for *trim*.

A síntese da máxima linguagem controlável de múltiplas restrições pode ser executada diretamente a partir das plantas locais sem aumento da restrição em relação à abordagem monolítica e, por conseguinte, à abordagem modular clássica, desde que a modularidade local seja verificada. Também o tamanho dos supervisores locais dependem apenas do tamanho da planta local, podendo ser pequeno mesmo quando o sistema a ser controlado é de grande porte.

É importante ressaltar que o teste de modularidade local, feito após a síntese dos supervisores para garantir a validade da abordagem, exige a composição de todos os supervisores locais. Assim, a complexidade da verificação da modularidade local cresce exponencialmente com o número de subsistemas e de restrições envolvidos.

2.3.3 Redução de Supervisores

Um supervisor minimamente restritivo incorpora informações redundantes, visto que ele contém restrições já realizadas pela planta. Isto significa que é possível reduzir o número de estados deste supervisor sem afetar sua ação de controle.

Diz-se que dois supervisores S_1 e S_2 são equivalentes se suas ações de controle sobre a planta P produzirem o mesmo comportamento, ou seja, se S_1/P for equivalente a S_2/P. Portanto, S_2 é um supervisor reduzido para S_1 se:

- O número de estados de S_2, denotado por $|S_2|$, for menor que $|S_1|$;
- $L(P) \cap L(S_2) = L(S_1)$;
- $L_m(P) \cap L_m(S_2) = L_m(S_1)$.

Dado um supervisor S, pode-se obter um supervisor reduzido S_{red} agregando-se os estados de $S\|P$ em blocos, não necessariamente disjuntos. No processo de agregação,

deve-se garantir que as desabilitações dos estados de um bloco não entrem em conflito com os eventos habilitados em outros estados do mesmo bloco e que o supervisor reduzido, representando a estrutura de transição entre os blocos, seja determinista. De forma a preservar a igualdade das linguagens marcadas, deve-se também garantir que estados pertencentes ao mesmo bloco estejam consistentemente marcados.

Um algoritmo de redução de supervisores é apresentado em [Vaz and Wonham, 1986]. Esse algoritmo tem complexidade exponencial no tempo e, devido a isso, é realizável apenas para autômatos de pequeno porte. Neste mesmo artigo, Vaz e Wonham provam que existe um tamanho mínimo de supervisor reduzido para um dado S, mas pode haver múltiplos supervisores com o número mínimo de estados.

A busca por supervisores suficientemente reduzidos através de algoritmos de menor complexidade computacional é válida. [Su and Wonham, 2003] apresentam um algoritmo de complexidade polinomial para a redução de supervisores baseado na agregação dos estados em blocos disjuntos, não computando, desta forma, um supervisor mínimo.

Uma documentação sobre a redução de supervisores mais detalhada é apresentada no Capítulo 4, onde o algoritmo implementado no Grail para Controle Supervisório está descrito.

2.4 Exemplos

Para elucidar as idéias apresentadas no decorrer deste capítulo, dois exemplos são abordados nesta seção. As composições síncronas necessárias a resolução dos exemplos, assim como a computação dos supervisores minimamente restritivos, foram feitos com o auxílio do ambiente Grail, apresentado no Capítulo 4.

2.4.1 O Gato e o Rato num Labirinto

Um gato e um rato compartilham o labirinto apresentado na Figura 2.2, sendo que as setas g_i sinalizam as passagens em mão única exclusivas para gatos e as setas r_i as passagens para o rato [Ramadge and Wonham, 1989]. Todas as passagens, com exceção da g_7, podem ser fechadas pelo controlador. O gato e o rato encontram-se inicialmente nos recintos indicados pelos respectivos ícones. Espera-se obter um controlador que garanta a maior liberdade de movimento evitando que ambos os animais ocupem o mesmo recinto ao mesmo tempo e que ambos possam sempre voltar ao estado inicial.

Modela-se o movimento do gato e do rato pelos autômatos P_g e P_r, respectivamente. Estes autômatos são apresentados na Figura 2.3. Importante salientar que os números

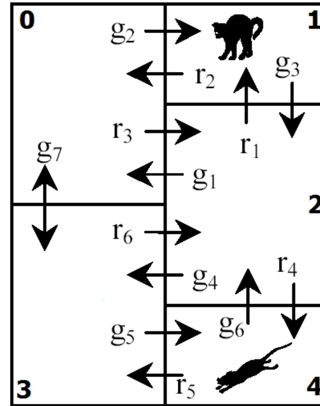


Figura 2.2: Labirinto do Gato e do Rato

associados aos estados de cada autômato, P_g e P_r, correspondem ao recinto em que o gato e o rato se encontram, respectivamente. A planta global pode ser obtida por $P = P_g \| P_r$.

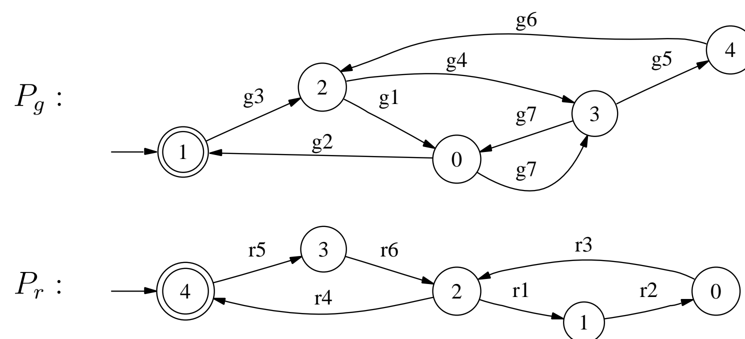


Figura 2.3: Modelos Obtidos para o Gato (P_g) e para o Rato (P_r)

O autômato $\widetilde{K}$, que representa a linguagem-alvo K, é obtido de P, apagando-se os estados em que os dois animais ocupem o mesmo recinto, e a máxima linguagem controlável contida em K, $supC(K, P)$, é então computada. Obteve-se o supervisor mostrado na Figura 2.4.

2.4.2 Célula de Manufatura

Uma célula de manufatura é composta por uma mesa circular de quatro posições (M_0), onde são efetuadas as operações de enchimento e tampamento de garrafas, e por mais quatro dispositivos operacionais: um alimentador (M_1), um enchedor (M_2), um tampador (M_3) e um manipulador robótico (M_4), como ilustrado na Figura 2.5 [Bouzon et al., 2004].

O programa de controle original da mesa permite operar em seqüência apenas uma garrafa por vez, ou seja, o alimentador só pode posicionar uma garrafa na mesa no-

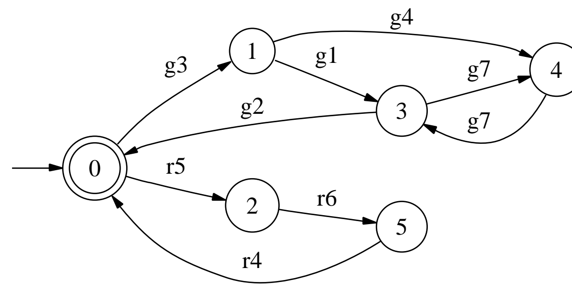


Figura 2.4: Supervisor Obtido para o Labirinto do Gato e do Rato

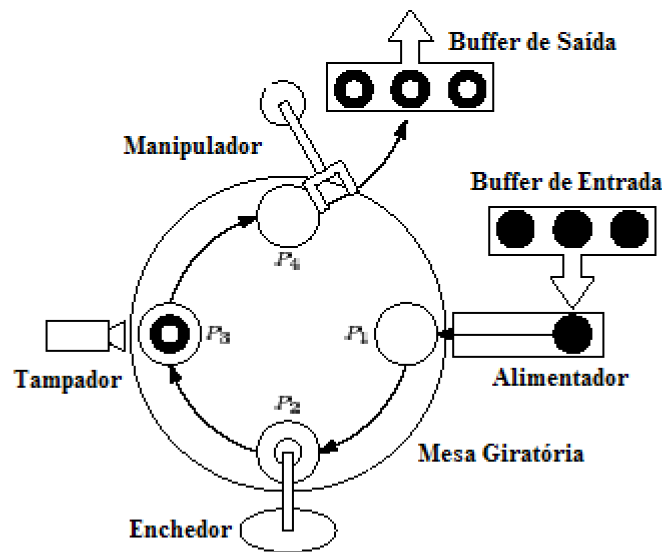


Figura 2.5: Mesa Giratória

vamente depois que o manipulador tiver retirado a garrafa anterior da mesa. Esta restrição da lógica de controle evita os problemas que podem ocorrer na operação de múltiplas garrafas em paralelo, sendo estes:

1. Giro da mesa sem a presença de garrafa na mesma;
2. Giro da mesa sem que as garrafas em P_2, P_3 e P_4 tenham sido enchidas, tampadas ou retiradas da mesa, respectivamente;
3. Giro da mesa enquanto o alimentador, o enchedor, o tampador ou o manipulador estiverem operando;
4. Operação do alimentador, do enchedor, do tampador ou do manipulador enquanto a mesa estiver girando;
5. Operação do alimentador, do enchedor, do tampador ou do manipulador sem que haja peças em P_2, P_3 e P_4, respectivamente;

6. Sobreposição de garrafas em P_1;

7. Enchimento ou tampamento da mesma garrafa duas vezes.

Esse modo de funcionamento - de uma garrafa por vez - é muito pouco eficiente, visto que o alimentador, o enchedor, o tampador e o manipulador passam a maior parte do tempo ociosos enquanto poderiam estar operando em paralelo.

O objetivo deste exemplo é, portanto, sintetizar um programa de controle que garanta uma maior eficiência da célula de manufatura, isto é, que permita a operação concorrente de 0 a 4 peças pelas 5 máquinas sem que ocorram os problemas já especificados.

Para a obtenção do modelo global que representa o funcionamento em malha aberta da célula, faz-se, inicialmente, a modelagem independente de cada dispositivo, em termos dos eventos de início e final de operação. Assim, considera-se que os subsistemas M_i, $i = 0, ..., 4$, têm suas operações iniciadas pelos eventos controláveis s_i e terminam de operar com o evento não-controlável f_i. As máquinas M_i podem, então, ser modeladas pelos autômatos P_i, representados na Figura 2.6.

$P_i, i = 0, ..., 4:$

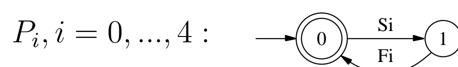


Figura 2.6: Autômatos para M_i, $i = 0, ..., 4$

A planta da célula de manufatura é, então, representada de forma monolítica por um autômato P obtido a partir da composição síncrona dos modelos dos cinco dispositivos, $P = \|_{i=0}^{4} P_i$.

As restrições a serem modeladas tem como objetivo evitar os problemas que ocorrem na operação de múltiplas garrafas em paralelo. Inicialmente, modela-se uma restrição que garanta que a mesa não vai girar à toa, isto é, sem ao menos uma garrafa vazia em P_1, uma garrafa cheia em P_2 ou uma garrafa tampada em P_3. De acordo com essa restrição, pelo menos um dos eventos f_1, f_2 e f_3, que representam respectivamente o fim de operação em P_1, P_2 e P_3, deve preceder a ocorrência do evento s_0, que inicia o giro da mesa. Essa restrição é modelada pelo autômato E_a, apresentada na Figura 2.7.

As quatro restrições de segurança que impedem a mesa de girar enquanto algum dos outros quatro dispositivos estiverem operando e vice-versa podem ser descritas pelo autômato E_{bi}, $i = 1, ..., 4$, da Figura 2.8. De forma genérica, a restrição E_{bi} garante que, quando a mesa M_0 ou algum dispositivo M_i começar a operar, os eventos s_0 e s_i não poderão mais ocorrer até que seja sinalizado o fim de operação.

Os possíveis problemas decorrentes do fluxo de múltiplas garrafas na mesa podem ser evitados pelas restrições: E_{c1}, relativa à movimentação de garrafas vazias entre as

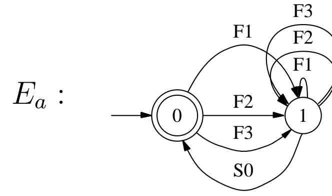


Figura 2.7: Autômatos para a Especificação E_a

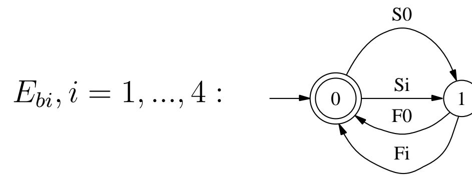


Figura 2.8: Autômatos para as Especificações E_{bi}, $i = 1, ..., 4$

posições P_1 e P_2; E_{c2}, para a manipulação de garrafas cheias entre P_2 e P_3; e E_{c3}, correspondente ao fluxo de garrafas cheias e tampadas entre P_3 e P_4. Para isso, a restrição E_{c1} evita sobrepor garrafas em P_1, encher sem garrafa vazia em P_2 e girar a mesa com garrafa vazia em P_2. Já E_{c2} proíbe encher duas vezes a mesma garrafa em P_2, tampar sem garrafa cheia em P_3 e girar a mesa com garrafa não-tampada em P_3. Enquanto que a restrição E_{c3} impede o tampamento de uma garrafa já tampada em P_3, o acionamento do manipulador sem garrafa em P_4 e girar a mesa com garrafa em P_4. Como as restrições têm a mesma estrutura, pode-se ilustrá-las pelo modelo indexado em i, E_{ci}, da Figura 2.9.

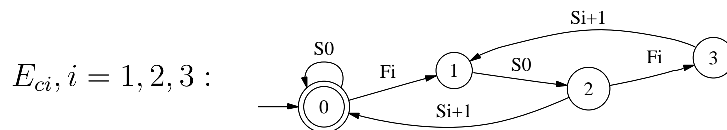


Figura 2.9: Autômatos para as Especificações E_{ci}, $i = 1, 2, 3$

A especificação monolítica E é calculada através da composição das restrições E_x, $x \in \{a, b_1, b_2, b_3, b_4, c_1, c_2, c_3\}$. Os próximos passos consistem no cálculo do comportamento desejado do sistema, $\widetilde{K} = E \| P$, e da máxima linguagem controlável contida em K, $supC(K, P)$.

Resolve-se este problema também por intermédio da abordagem modular local. Neste caso, deve-se obter as plantas locais P_x respectivas às restrições E_x fazendo-se a composição dos modelos de P_i, $i = 1, ..., 4$, que tenham eventos comuns a elas. Formam-se, desta forma, os seguintes autômatos:

- $P_a = P_0 \| P_1 \| P_2 \| P_3$;

- $P_{bi} = P_0 \| P_i, i = 1, ..., 4$;
- $P_{ci} = P_0 \| P_i \| P_{i+1}, i = 1, 2, 3$.

Calculam-se, então, os comportamentos desejados locais $\widetilde{K_x}$, $x \in \{a, b_1, b_2, b_3, b_4, c_1, c_2, c_3\}$, pela composição síncrona das restrições E_x com suas respectivas plantas locais P_x. Os supervisores locais são, então, computados e a modularidade local entre eles é verificada. A resolução deste exemplo é mostrada no Capítulo 4.

2.5 Conclusões

Este capítulo apresentou, baseado em [Cury, 2001], os conceitos básicos da Teoria de Controle Supervisório incluindo linguagens, expressões regulares e autômatos. O próximo capítulo introduz um ambiente computacional que manipula estes, servindo, portanto, como uma ferramenta de auxílio para a Teoria de Controle Supervisório apresentada por Ramadge e Wonham e suas extensões. O ambiente Grail é descrito desde sua última versão original, a 2.5, até as contribuições fornecidas por este trabalho.

Capítulo 3

O Grail e sua Nova Estrutura

O Grail é um ambiente de computação simbólica para pesquisa, ensino e aplicações que envolvem Linguagens Finitas (FLs), Expressões Regulares (REs) e Máquinas de Estados Finitos (FMs). Ele consiste de uma coleção de programas que operam sobre arquivos de entrada, cujo conteúdo é uma descrição textual de seus objetos de manipulação.

A implementação do Grail foi iniciada por Darrell Raymond [Raymond, 2005] e Derick Wood [Wood, 2005] no Departamento de Ciências da Computação da Universidade de *Western Ontario*, Canadá [Raymond and Wood, 1996a]. O projeto [Raymond and Wood, 2002] encontra-se estagnado desde 2002, sendo sua última versão estável a 2.5, apresentada na Seção 3.1.

O Grail versão 2.5 vem sendo expandido, sob a orientação de José Eduardo Ribeiro Cury [Cury, 2005], no Departamento de Automação e Sistemas da Universidade Federal de Santa Catarina, como documentado na Seção 3.2. O objetivo principal desta expansão tem sido focar a utilização deste ambiente para a Teoria de Controle Supervisório.

Com o propósito de controlar esta expansão, que vem ocorrendo de forma descontrolada, este trabalho visa uma reestruturação, e conseqüentemente documentação deste ambiente, e o resultado disso é o Grail para Controle Supervisório, introduzido na Seção 3.3.

3.1 O Grail Versão 2.5

3.1.1 Conceitos Básicos

Os três objetos principais do Grail são as FLs, as REs e as FMs. Sendo que estes objetos são recebidos, e também retornados, pelos filtros do Grail como uma descrição textual ASCII que podem ser geradas a partir de qualquer editor de texto.

Uma FL é um conjunto finito de palavras de comprimento finito. Um exemplo de FL é mostrado abaixo.

```
% type fl1
abc
aabc
babc
```

As REs seguem a notação convencional e suportam a união, a concatenação e o fechamento Kleene. São exemplos de REs: `a+b`, `((a+bcde*)+c)*`, {} e `''+a`. O Grail segue também as regras normais de precedência, sendo a do fechamento Kleene a maior e a da união, a menor [Raymond and Wood, 1996d].

Uma FM é definida por uma tripla $FM = (Q_i, \Theta, Q_f)$, onde Q_i é o conjunto de estados iniciais, Θ é o conjunto de instruções e Q_f é o conjunto de estados finais. Cada instrução $\theta \in \Theta$ é composta por:

- Um estado fonte θ_{source}, a partir do qual uma transição sai;
- Uma etiqueta de transição θ_{event};
- Um estado destino θ_{sink}, aonde a transição em questão chega.

Um autômato $A = (Q, \Sigma, \delta, q_0, Q_m)$ é representado por uma $FM = (Q_i, \Theta, Q_f)$, tal que:

- $Q = Q_i \cup \{(\theta_{source} \cup \theta_{sink}), \forall\theta \in \Theta\} \cup Q_f$;
- $\Sigma = \{\theta_{event}, \forall\theta \in \Theta\}$;
- $(\forall\theta \in \Theta)\delta(\theta_{source}, \theta_{event}) = \theta_{sink}$;
- $q_0 = \{q \in Q_i\}$;
- $Q_m = Q_f$.

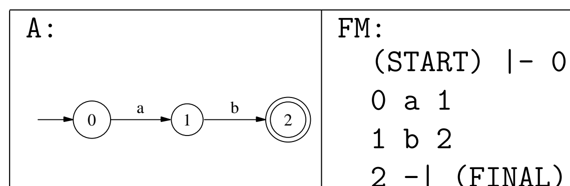


Figura 3.1: Representação de um Autômato A por uma FM

Um exemplo é apresentado na Figura 3.5. Note que os estados iniciais e os finais de uma FM são indicados através de instruções que possuem como estado fonte e destino os pseudo-estados `(START)` e `(FINAL)`, respectivamente.

No Grail, todos os estados são designados por números naturais. Já o alfabeto de uma FM é dado pelos símbolos que aparecem nas instruções, a menos das pseudo-etiquetas `|-` e `-|`. O tipo de alfabeto é parametrizável, podendo este ser constituído por caracteres, a exemplo da Figura 3.1, seqüências de caracteres ou, até mesmo, REs. Os algoritmos do Grail são independentes do tipo de alfabeto estabelecido.

Os filtros do Grail são acessíveis em linha de comando. Utiliza-se qualquer janela de terminal do sistema para acessá-los, incluindo janelas *telnet* e *ssh*. Os objetos a serem utilizados como parâmetros podem ser direcionados diretamente a partir do teclado ou redirecionados a partir de arquivos. Por exemplo, a concatenação de duas FMs, `fm1` e `fm2`, pode ser computado da seguinte forma.

```
% fmcat fm1 fm2 > fm12
```

Neste caso, o resultado está sendo redirecionado para o arquivo `fm12`. Se este arquivo já existir, ele será sobrescrito sem confirmação. Se a saída não for redirecionada, o resultado é impresso no próprio terminal.

A saída de um filtro pode ser utilizada como primeiro parâmetro para um segundo filtro. Por exemplo, para se computar a concatenação de três FMs, `fm1`, `fm2` e `fm3`, utiliza-se o seguinte comando.

```
% fmcat fm1 fm2 | fmcat fm3 > fm123
```

Os operadores | e > são suportados por qualquer terminal do sistema UNIX assim como pelo MS-DOS.

A instalação do Grail é feita manualmente. Para isso, deve-se baixá-lo e descompactá-lo para o diretório desejado. Além disso, é necessário inserir o caminho `[grail_dir]\bin` no *path* do sistema, já que é ali que os filtros do Grail estão armazenados.

3.1.2 Organização dos Arquivos

O Grail é um pacote organizado nos seguintes diretórios [Raymond and Wood, 1996b]:

- **bin:** possui os filtros do Grail para um dado alfabeto;
- **binaries:** contém o código do Grail compilado para os tipos de alfabeto definidos;
- **classes:** armazena as classes que definem os objetos que o Grail pode manipular;
- **doc:** documentação;
- **man:** documentação *online*;
- **tests:** composta por máquinas de teste, ou seja, exemplos resolvidos sobre os quais os filtros podem ser testados.

3.1.3 Implementação

O Grail é implementado em C++, já que esta linguagem encoraja o encapsulamento e a definição de interfaces [Raymond and Wood, 1996b]. Como o Grail foi criado com o objetivo de ser continuamente expandido, este preza pela reutilização de código facilitada pela programação orientada a objetos, característico da linguagem C++, e pelo uso em larga escala de gabaritos. Gabaritos são um dos recursos mais poderosos de C++ para a reutilização de código [Deitel and Deitel, 2001]. Entretanto, eles devem ser usados com cautela, pois o seu poder de instanciação é proporcional ao aumento da complexidade do código.

A programação orientada a objetos encapsula dados, aqui chamados atributos, e funções que manipulam este dados, denominadas métodos, em pacotes chamados classes. A partir de uma classe, um programador pode criar um ou mais objetos [Deitel and Deitel, 2001]. O Grail versão 2.5 possui 18 classes, sendo que praticamente todas elas são gabaritos de classes.

Os gabaritos são mecanismos de abstração existentes em C++ que fornecem suporte para a programação genérica e flexível. Gabaritos de classes são chamados de tipos parametrizados porque exigem um ou mais parâmetros de tipo a especificar. A especificação deste(s) parâmetro(s) personaliza um gabarito de uma *classe genérica* formando uma classe gabarito específica. O programador que deseja usar classes gabarito escreve um gabarito de classe. Quando o programador necessita de uma nova classe para um tipo específico, ele usa uma notação concisa e o compilador escreve o código-fonte para a classe gabarito [Deitel and Deitel, 2001]. Em outras palavras, os gabaritos

definem classes como se estas fossem caixas pretas prontas para serem instanciadas de acordo com a escolha dos parâmetros do programador [Stroustrup, 2000].

As classes do Grail versão 2.5 são: `FL`, `RE`, `FM`, `State`, `Inst`, `Subexp`, `Empty_set`, `Empty_string`, `Symbol_exp`, `Cat_exp`, `Plus_exp`, `Star_exp`, `Array`, `Set`, `List`, `String`, `Bits` e `Pool`. Suas classes principais, que definem seus objetos de manipulação, são: `FL` (Linguagem Finita), `RE` (Expressão Regular) e `FM` (Máquina de Estados Finitos).

As classes secundárias podem ser divididas em dois grupos. O primeiro define estruturas básicas, sendo elas `Array`, `String`, `List`, `Set`, `Pool` e `Bits`. A classe `Bits` gerencia *bitmaps* e `Pool` gerencia memória alocada. `Set`, `List` e `String` são classes que herdam de `Array` seus atributos e métodos [Raymond and Wood, 1996b]. A herança é uma forma de reutilização de código na qual novas classes são criadas a partir de classes existentes pela absorção de seus atributos e métodos e redefinindo ou *embelezando-as* com recursos que as novas classes requerem. A nova classe é chamada de uma classe derivada. Cada classe derivada se torna, ela própria, uma candidata a ser uma classe base para uma futura classe derivada [Deitel and Deitel, 2001].

O segundo grupo caracteriza as sub-estruturas das classes principais, que são `State`, `Inst` e `Subexp`. Esta última, `Subexp`, é uma classe abstrata, cujo único objetivo é fornecer uma classe base apropriada a partir da qual outras classes podem herdar a interface e/ou a implementação. Nenhum objeto desta classe pode ser instanciado [Deitel and Deitel, 2001]. Suas classes derivadas são: `Empty_set`, `Empty_string`, `Symbol_exp`, `Cat_exp`, `Plus_exp` e `Star_exp`. Uma descrição mais detalhada das classes do Grail é encontrada na próxima seção.

3.1.3.1 Classes

Esta seção apresenta as classes do Grail versão 2.5 detalhadamente. Alguns diagramas de classes foram construídos no decorrer deste trabalho para um melhor entendimento do relacionamento existente entre as mesmas. Foi utilizado o padrão UML (*Unified Modelling Language*)[Fowler, 2003] para a construção destes diagramas. Um diagrama de classes descreve os tipos de objetos presentes no sistema e os vários tipos de relacionamentos estáticos existentes entre eles. Ele também mostra os atributos e os métodos de cada classe, assim como as limitações aplicadas as formas com as quais os objetos estão conectados.

Os diagramas de classes construídos não mostram os métodos de cada classe, pois isso os tornaria um tanto quanto ilegíveis. Quanto aos atributos, convém citar que os símbolos # e − presentes antes de cada declaração indicam se o atributo é protegido[1]

[1]Pode ser acessado apenas por membros e `friends` da classe base e de suas classes derivadas.

ou privado[2], respectivamente. O símbolo + indicaria que o atributo é público[3], porém nenhuma classe do Grail possui atributos públicos. Algumas classes possuem atributos de tipo a especificar, o que indica que são gabaritos de classes. Estas classes possuem uma caixa de anotação no seu canto superior e o parâmetro a especificar está escrito dentro desta anotação [Fowler, 2003].

As classes existentes no Grail versão 2.5 são, portanto:

- **FL:** Gabarito de classe que implementa as linguagens finitas, compostas por um conjunto finito de palavras de comprimento finito. Possui apenas um atributo, sendo este um conjunto, cujos elementos são seqüências comparáveis de algo a ser instanciado `Item`.

  Teoricamente, o parâmetro `Item` pode ser instanciado como caracteres, seqüências de caracteres ou expressões regulares. Entretanto, o modo com que o Grail diferencia as formas de abstração para a classe `FL` não está bem definido, sendo esta funcional apenas quando a classe é instanciada em caracteres.

  A Figura 3.2 mostra o diagrama de classes cuja classe principal é `FL`.

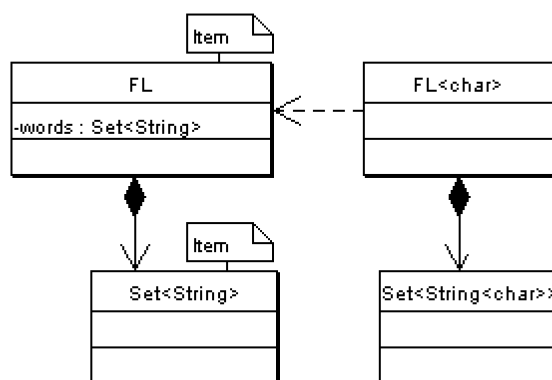


Figura 3.2: Diagrama de Classes `FL`

  Tanto os conjuntos (`Set`) como as seqüências comparáveis (`Strings`) que estruturam `FL` são definidos como classes derivadas de uma outra classe do Grail, a `Array`.

- **Array:** Gabarito de classe que define uma estrutura de dados básica. Seus objetos são vetores cujos elementos podem ser caracteres, sequências de caracteres, expressões regulares, estados, instruções, máquinas de estados finitas e assim por diante.

  `Array` possui três classes derivadas, que são:

[2]Pode ser acessado apenas por membros e `friends` da classe base.
[3]Pode ser acessado por todas as funções do programa.

- **Set:** define conjuntos que não são ordenados e que não possuem elementos duplicados.
- **List:** cujos objetos são conjuntos ordenados que possuem elementos duplicados.
- **String:** define conjuntos ordenados que possuem elementos duplicados e que podem ser comparados entre si.

A Figura 3.3 mostra o diagrama de classes cuja classe principal é `Array`. Esta classe possui vários atributos, com destaque a um deles, o `Item`. `Item` é um tipo a especificar e pode ser instanciado de acordo com os parâmetros do programador. `Item` é o parâmetro que caracteriza `Array` como um gabarito de classe.

O diagrama de classes `Array` mostra as três classes derivadas da mesma: `List`, `Set` e `String`. Algumas instâncias de `Set` são mostradas como exemplo. Observe também que `String` não é uma classe derivada diretamente de `Array`, e sim de uma instância desta. Em outras palavras, `String` não pode ser instanciada com a mesma variedade de tipos que `Array`.

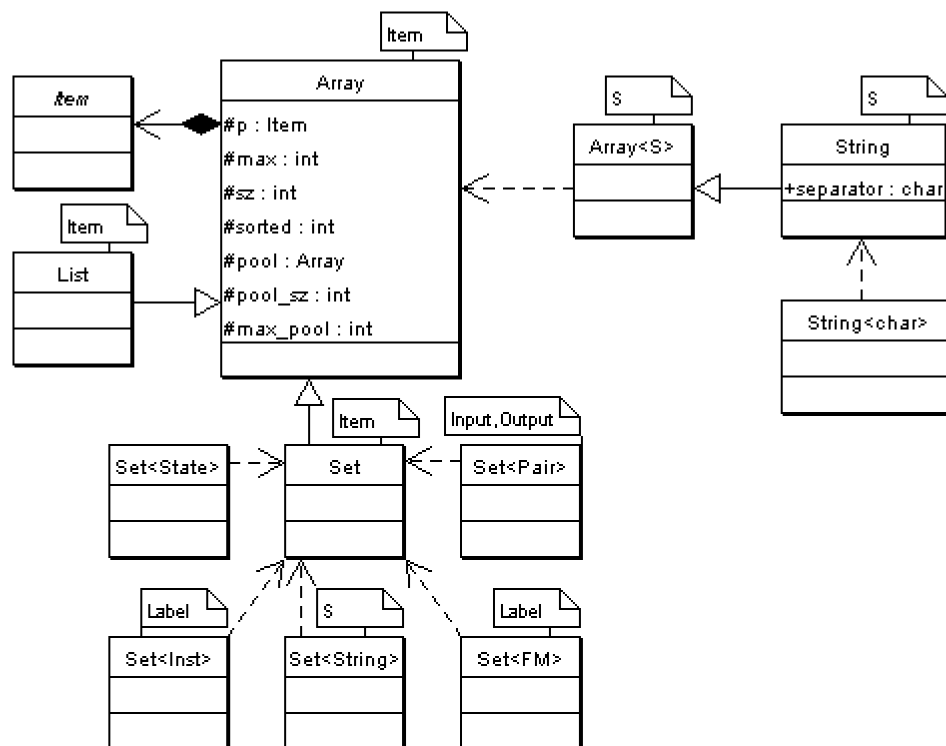


Figura 3.3: Diagrama de Classes `Array`

- **RE:** Gabarito de classe que implementa as expressões regulares, as estruturas mais complexas do Grail. Como mostra a Figura 3.4, a classe `RE` é composta por uma subexpressão, sendo que a classe que caracteriza esta subexpressão, `Subexp`, é abstrata e serve apenas como base para suas seis classes derivadas,

todas gabaritos de classe. A subexpressão de uma `RE` é, portanto, composta por um agrupamento ordenado de objetos das classes derivadas de `Subexp`. Por exemplo, a expressão $(a^* + bc)$ é dada por:

```
Plus_exp<x,y>, onde: x=Star_exp<Symbol_exp<char>>
                     y=Cat_exp<Symbol_exp<char>,Symbol_exp<char>>
```

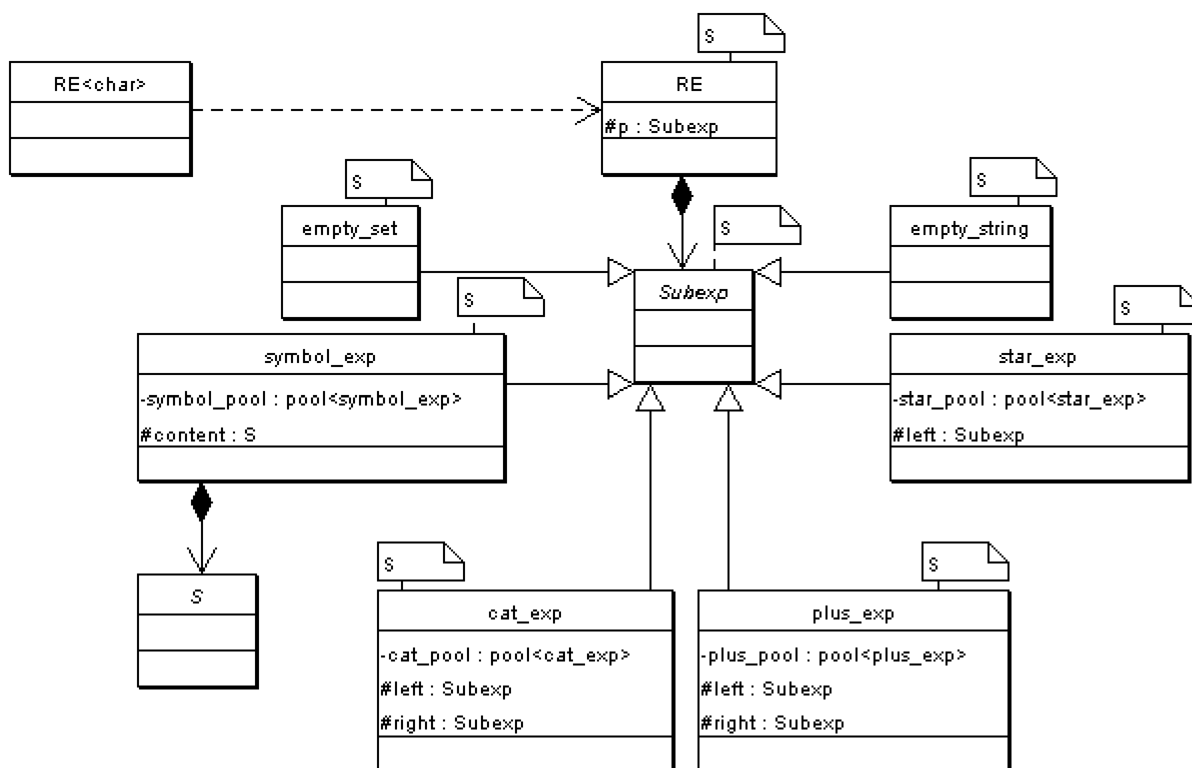


Figura 3.4: Diagrama de Classes `RE`

- **Subexp:** Gabarito de classe abstrata que serve como base para o conjunto de possíveis subexpressões. Cada tipo de subexpressão caracteriza um gabarito de classe derivada, sendo eles:
  - **Empty_set:** representa o conjunto vazio $\{\}$.
  - **Empty_string:** caracteriza a seqüência de caracteres vazia ' '.
  - **Symbol_exp:** define um símbolo simples, presente em um dos intervalos `a-z` ou `A-Z`. Possui apenas um atributo, que é o próprio símbolo em questão.
  - **Cat_exp:** implementa a concatenação `xy`. Possui dois atributos, `x` e `y`, sendo que estes parâmetros representam outras subexpressões.
  - **Plus_exp:** representa a união $x + y$. Possui dois atributos, `x` e `y`, sendo que estes parâmetros representam outras subexpressões.
  - **Star_exp:** define o fechamento Kleene x^*. Possui apenas um atributo, `x`, sendo que este parâmetro representa outra subexpressão.

- **FM:** Gabarito de classe que implementa as máquinas de estados finitos. Possui três atributos, como mostra a Figura 3.5, sendo estes `Set`s de objetos de outras classes do Grail. São elas:

  – `State`, cujos objetos compõem os conjuntos de estados iniciais e finais;

  – `Inst`, cujos objetos representam as instruções da FM.

  `FM` pode ser instanciada de acordo com o alfabeto desejado, seja ele composto por caracteres, seqüências de caracteres e, até mesmo, expressões regulares.

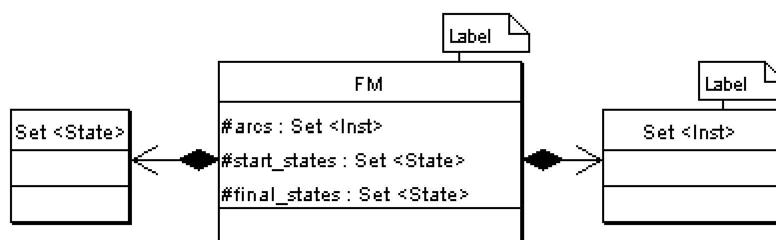


Figura 3.5: Diagrama da Classes `FM`

- **State:** Implementa os estados de uma máquina de estados finitos. Os estados são representados por números inteiros não-negativos acrescidos, internamente, de 2, já que os números 0 e 1 representam os pseudo-estados START e FINAL, respectivamente.

- **Inst:** Gabarito de classe que implementa as instruções de uma máquina de estados finitos. Possui três atributos, que são `source` (θ_{source}), `label` (θ_{event}) e `sink` (θ_{sink}). Os atributos `source` e `sink` são objetos da classe `State`, já `label` é um parâmetro de tipo a especificar e será especificado de acordo com o tipo de alfabeto utilizado pela FM.

  A Figura 3.6 representa o diagrama de classes `Inst`, aonde estão presentes alguns exemplos de instanciação desta.

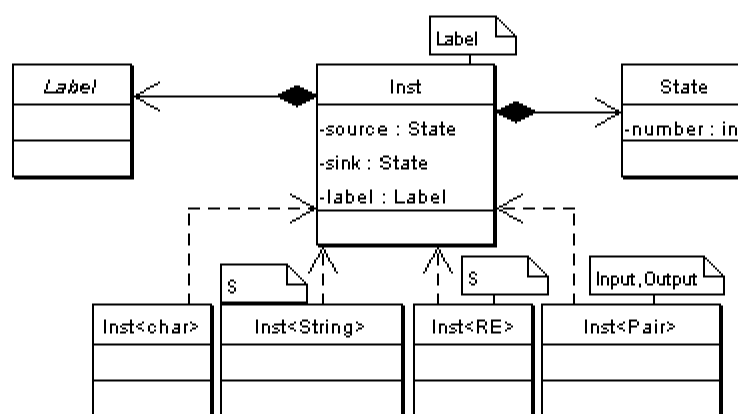


Figura 3.6: Diagrama de Classes `Inst`

- **Bits:** Classe que gerencia *bitmaps*. *Bitmaps* são um tipo de uma tabela, de tamanho a ser definido, cujos elementos podem ser setados em 0 ou 1.

- **Pool:** Gabarito de classe que gerencia memórias dinâmicas de propósito geral para classes que possuem um grande número de pequenos objetos, como a `RE` e a `FM`. O objetivo principal desta classe é eliminar o tempo desperdiçado pelo Grail na criação e destruição de *arrays* temporários. Afinal, programas em C++ podem ser melhorados significativamente utilizando alocação de memória sob encomenda ao invés da criação e destruição de *arrays* [Raymond and Wood, 1996c].

  A classe `Pool` faz com que o Grail aloque uma certa quantidade de memória pré-definida na criação de um objeto ao invés de alocar apenas um *array* do tamanho necessário. A partir daí, em vez de ficar alocando e liberando memória, o Grail utiliza os elementos do *array* já alocado enquanto a classe `Pool` gerencia estes elementos utilizando um *bitmap* para verificar quais estão em uso. Apenas quando toda a memória alocada estiver em uso é que mais memória é alocada, observando que o tamanho do próximo *Array* será o dobro do anterior.

  Desta forma, as funções *new* e *delete* são muito mais rápidas que as *default* pois esta classe simplesmente gerencia apontadores para memória existente. A classe `Pool` permite a reutilização de memória já retornada, possuindo alta fragmentação e baixo *overhead* por não rearranjar os objetos.

  Esta classe foi criada para a versão 2.5 do Grail e tem como finalidade aumentar a eficiência de seus filtros, tornando-os mais rápidos. Entretanto, foi implementada apenas para a classe `RE`, como mostra a Figura 3.7 - todas as instâncias possíveis de `Pool` estão ali presentes. Uma extensão desta classe para a `FM` ficou como perspectiva para a versão 3.0, que nunca chegou a ser concluída.

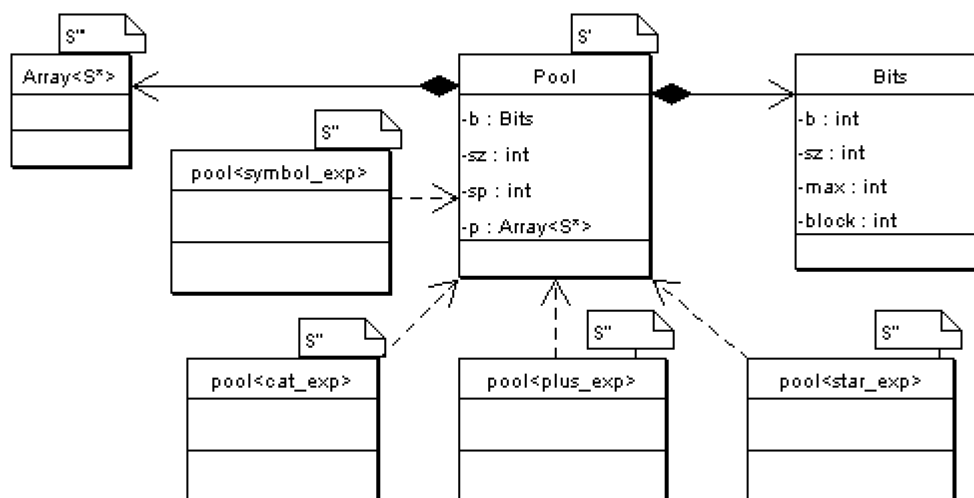


Figura 3.7: Diagrama da Classes `Pool`

3.1.3.2 A Estrutura do Grail Versão 2.5

A exemplo da Figura 3.8, esta seção apresenta a forma como o código fonte do Grail está estruturado a partir do seu nível mais baixo, as classes, até o mais alto, o Grail em si. Toda classe do Grail possui os seguintes arquivos:

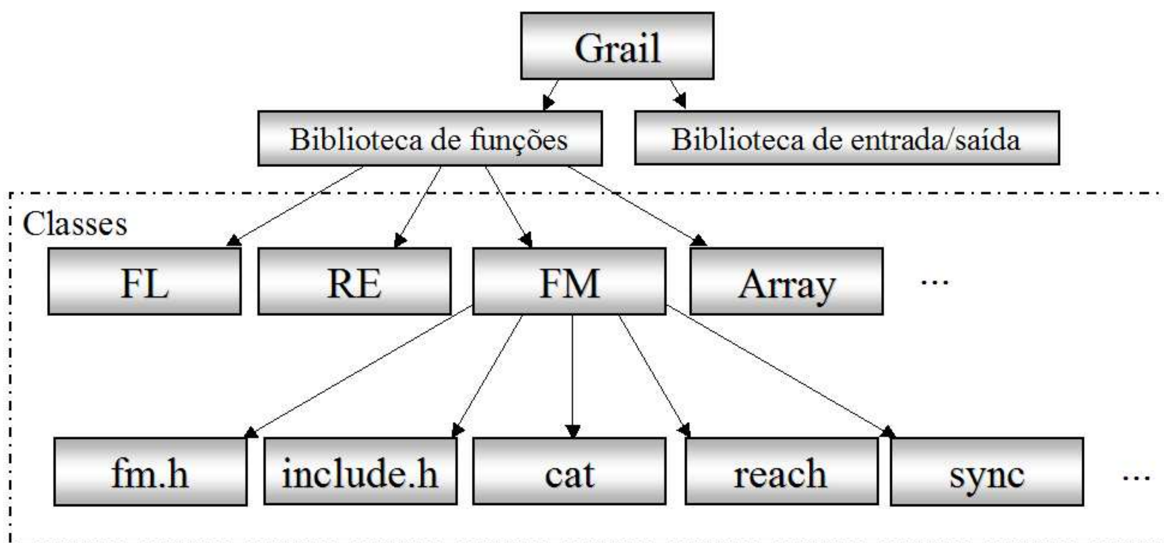


Figura 3.8: Estrutura da Grail Versão 2.5

- Um arquivo de descrição que contém a declaração de todos os métodos da classe, denominado `nome_da_classe.h`;
- Vários arquivos `nome_do_método.src`, aonde estão contidas as implementações dos métodos. A extensão `src` possibilita a concatenação de todos os algoritmos em um arquivo de descrição da classe em questão [Raymond and Wood, 1996b];
- Duas funções `friend`, armazenadas também em arquivos `src`, que definem os operadores de entrada e saída, >> e <<, respectivamente[4];
- Um arquivo contendo todos os arquivos da classe que serão incluídos na compilação, denominado `include.h`.

Os arquivos `include.h` de todas as classes estão listados no `grail.h`, presente na pasta `binaries` da versão 2.5. Este arquivo é incluído no `grail.cpp` como uma

[4]Uma função `friend` de uma classe é definida fora do escopo daquela classe, mas ainda tem o direito de acessar os membros privados e protegidos da classe. Quando uma função operador é implementada como um método, o operando mais à esquerda deve ser um objeto da classe do operador. Se o operando esquerdo for um objeto de uma classe diferente ou um tipo primitivo, esta função de operador não deve ser implementada como um método. Entretanto, se esta deve acessar diretamente membros privados ou protegidos daquela classe, ela deve ser definida como uma função `friend`. O operador << deve ter um operando à esquerda do tipo `ostream&`. De modo semelhante, o operador sobrecarregado >> deve ter um operando à esquerda do tipo `istream&` [Deitel and Deitel, 2001].

biblioteca de funções. No `grail.cpp` está presente a função `main` deste ambiente, assim como as funções de recebimento de parâmetros.

O função `main` do Grail consiste, basicamente, de uma lista de `ifs`, aonde o executável verifica sob qual nome foi chamado e realiza a operação referente a este nome. Na realidade, todos os filtros do Grail versão 2.5 são o `grail.exe` renomeado e copiado para a pasta `bin`.

3.2 Evolução do Ambiente Grail

O Grail versão 2.5 vem sendo expandido com o objetivo de focar a utilização deste ambiente na Teoria de Controle Supervisório. O objeto base desta teoria são os autômatos, conseqüentemente, as FMs do Grail. Devido a isso, as classes referentes às FLs e as REs foram desabilitadas no decorrer desta evolução.

Filtros referentes ao Controle Supervisório básico, descritos no Capítulo 4, foram implementados, alterando, desta forma, a classe `FM`. Um exemplo deste é a criação do filtro do Grail que converte as FMs para o formato aceito pelo Graphviz. O Graphviz é uma ferramenta de visualização encontrada em `http://www.research.att.com/sw/tool/graphviz`. Sua instalação é realizada por intermédio da inserção do caminho `[graphviz_dir]\att\graphviz\bin` no *path* do sistema.

Além disso, filtros referentes a Sistemas Condição/Evento [Rodrigues, 2004] e SEDs com Marcação Flexível [da Cunha, 2003] foram criados utilizando o Grail como uma biblioteca de funções. Entretanto, os Sistemas Condição/Evento utilizavam como base as classes `Pair` e `Iots`, que não estão presentes no Grail original.

A classe `Pair` define a estrutura de um par ordenado `[int,int]`, a ser usado como alfabeto das FMs no tratamento de Sistemas Condição/Evento. O primeiro `int` representa o evento e o segundo, a condição. A classe `Iots` é uma classe derivada da `FM`, cujo alfabeto é formado por objetos da classe `Pair`.

O resultado desta expansão foi um Grail base diferente do que consta na documentação da versão 2.5 e cuja documentação era inexistente ou estava espalhada nas diversas dissertações. Um Grail que possuía métodos que não estavam mais sendo usados ou que foram criados para serem utilizados por filtros que não estavam presentes no seu pacote. Pacotes independentes com filtros e um Grail, cujas classes também não eram idênticas às do Grail base.

Um dos objetivos principais deste trabalho é, devido a esta expansão descontrolada do Grail, a reorganização e a definição de uma estrutura mais adequada à expansão do mesmo. O resultado disso é o Grail para Controle Supervisório, introduzido na Seção 3.3.

3.3 A Nova Estrutura do Grail

A idéia da reestruturação do Grail é manter uma versão do Grail para Controle Supervisório Base, aonde permanecem os filtros provenientes da versão 2.5 e os que lidam com a Teoria de Controle Supervisório básica, apresentada no Capítulo 2. Os filtros já existentes e os que vierem a ser implementados que se relacionarem com expansões distintas da teoria básica serão acrescidos ao Grail como módulos, Seção 3.3.2, cuja estrutura é a mesma do Grail para Controle Supervisório e cujo controle também deverá ser feito através de versões. Esta é a forma mais modular encontrada para a expansão do Grail para Controle Supervisório, apesar de que a manutenção de versões de um ambiente de código aberto nem sempre é trivial.

A Seção 3.3.1 apresenta as principais modificações feitas na estrutura interna do Grail, com o objetivo de aumentar sua modularidade, e a Seção 3.3.3 mostra as regras de expansão do Grail para Controle Supervisório que visam uma expansão, no mínimo, organizada.

3.3.1 Principais Modificações

A principal modificação do Grail é a eliminação do `grail.cpp`. A idéia de apenas uma função `main` para todos os filtros tornava o Grail menos modular, pela existência de um ponto em comum desnecessário entre todos os filtros.

Conseqüentemente, o Grail para Controle Supervisório possui um arquivo `cpp` para cada filtro. Desta forma, cada filtro possui a sua função `main` e o seu executável específico. A implementação das funções de entrada e saída de parâmetros foram passadas para um `cpp` denominado `io.cpp` e a declaração destas funções estão presentes na biblioteca de entrada e saída `io.h`, acrescida ao `grail.h`.

No arquivo `cpp` de cada filtro do Grail foi acrescido um *help*, cujo objetivo é fornecer um texto de ajuda ao usuário sempre que este desejar. Para acessar este *help*, basta chamar o filtro com o parâmetro `-h`. O exemplo abaixo mostra o acesso ao *help* do filtro `fmcat`, utilizado em exemplos anteriores.

```
% fmcat -h
 fmcat: catenates two FMs.
 Syntax : fmcat FM1 FM2
 Version: May 1993
```

Todos os filtros ainda possuem um ponto em comum, que é a declaração de seus métodos dentro das classes.

A organização dos arquivos dentro do pacote Grail para Controle Supervisório, assim como uma discussão sobre as classes que permaneceram neste ambiente, são apresentadas nas seções seguintes.

3.3.1.1 Organização dos Arquivos

O Grail para Controle Supervisório é um pacote organizado nos seguintes diretórios:

- **bin:** contém os filtros propriamente ditos;
- **classes:** armazena as classes que definem os objetos que o Grail pode manipular;
- **doc:** documentação;
- **filters:** possui o código dos filtros, a biblioteca de classes `grail.h` e a biblioteca de entrada/saída `io.h`, assim como o arquivo com a implementação destas funções `io.cpp`.

3.3.1.2 Classes

Praticamente todas as classes do Grail versão 2.5 foram mantidas, a exceção da `FL`, que foi completamente eliminada por não ser útil à Teoria de Controle Supervisório. A classe `RE`, assim como sua sub-estrutura `Subexp` e derivadas, foram recuperados. Mas nem todos os filtros da classe `RE` permaneceram no Grail para Controle Supervisório.

Quanto as classes criadas durante a expansão do Grail, a classe `Pair` foi mantida por servir como uma estrutura básica, a exemplo da classe `Array`. Já `Iots` foi transferida para o módulo Sistema Condição/Evento.

3.3.2 Módulos

Os módulos são pacotes que possuem novos filtros que podem ou não utilizar novas classes, conseqüentemente novos métodos, e que utilizam como base as classes do Grail para Controle Supervisório. Estes pacotes podem ou não ser adicionados ao Grail base, dependendo do interesse do usuário no módulo, ou seja, na extensão da teoria em questão. Essa opção implica numa redução do espaço necessário à instalação do Grail, que pode crescer indefinidamente sem que todos os usuários necessitem ter todos os filtros.

A exemplo do Grail, os módulos também devem ser instaladas manualmente. Deve-se baixá-las, descompactá-las dentro do diretório do Grail e inserir o caminho `[grail_dir]\nome_do_modulo\bin` no *path* do sistema.

A estrutura do módulo assim como a organização dos arquivos seguem o padrão do Grail. Portanto, os módulos são organizados nos seguintes diretórios:

- **bin:** contém filtros propriamente ditos;
- **classes:** contém as classes que definem os objetos que o módulo pode manipular além das classes básicas;
- **doc:** documentação;
- **filters:** possui o código dos filtros, a biblioteca de classes `nome_do_modulo.h` e a biblioteca de entrada/saída do próprio módulo `io.h`, quando existente, assim como a implementação de suas funções em `io.cpp`.

Os módulos hoje existentes são:

- Sistemas Condição/Evento (Capítulo 7), cuja estrutura foi implementada no decorrer deste trabalho;
- Controle de SEDs com Marcação Flexível (Capítulo 6), cujas classes foram desenvolvidas no decorrer deste trabalho;
- Controle Multitarefa (Capítulo 5), totalmente implementada no decorrer deste trabalho.

3.3.3 Extensão do Grail para Controle Supervisório

A nova estrutura do Grail é apresentada na Figura 3.9. Com base nela, podemos citar as seguintes formas de expansão do Grail:

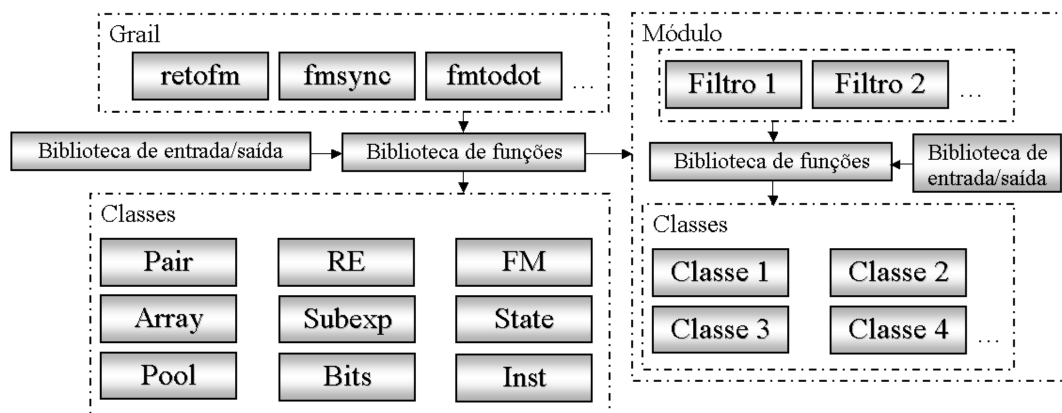


Figura 3.9: Estrutura do Grail para Controle Supervisório

- Acréscimo de um novo método, Seção 3.3.3.1;
- Adição de um novo filtro, Seção 3.3.3.2;
- Criação de um novo módulo, Seção 3.3.3.3.

3.3.3.1 Acréscimo de um Novo Método

O acréscimo de um novo método no Grail consiste nas etapas descritas abaixo. Um método, que recebe um parâmetro do tipo `FM` denominado `in`, será acrescido a classe `FM` como exemplo.

1. Escrever o algoritmo como um membro da classe desejada.

```
% type "classes/fm/example.src"
 template <class Label>
 void
 fm<Label>::example(fm<Label>& in) {
    // Algoritmo que implementa example
 }
```

2. Assegurar que o algoritmo será incluído no Grail, o que é feito em dois passos:

   (a) Adição do arquivo `example.src` no `include.h` da classe em questão, no caso `fm`.

   ```
   # include 'fm.h'
   ..
   # include 'example.src'
   ..
   ```

   (b) Declaração do método `example` na biblioteca de funções da classe em questão, no caso `fm.h`.

   ```
   template <class Label>
   class fm {
       protected:  \\ Atributos
           ..
       public:     \\ Métodos
           ..
           void example(fm<Label>& in);
           ..
   }
   ```

3.3.3.2 Adição de um Novo Filtro

A adição de um novo filtro é realizada após o acréscimo de um método que o implementa ou através da reunião de métodos já existentes, cujo seqüenciamento produza um resultado interessante. A adição de um novo filtro é realizada por intermédio dos seguintes passos:

1. Criação de um arquivo `cpp` que utilize o `grail.h` como biblioteca e tenha uma função `main`. A função `main` deve: possuir um *help* a ser chamado pelo usuário, receber os parâmetros, chamar o método implementado e determinar a saída. Um modelo para o `cpp` de um novo filtro é mostrado abaixo.

```
// fmexample.cpp
// Descrever o que este filtro faz...
//      Syntax : fmexample FM
// Version : September 2005

// Define compilers
#define MSVC

// Load libraries
#include "grail.h"

// Function main
int main(int argc,char** argv) {
    // Get my name
    char*       my_name;
    my_name = get_my_name(argv);

    // Get function's help
    char help_text[500];
    strcpy(help_text,my_name);
    strcat(help_text,": Descrever o que este filtro faz...\n");
    strcat(help_text,"Syntax : fmexample FM\n");
    strcat(help_text,"Version: September 2005\n");

    // Variables
    fm<string<char> >   in,out;
    // Get parameters
    get_one(in, argc, argv, my_name, help_text);
```

```
        // Do the operation
        out.example(in);
        cout << out;
        return 0;
    }
```

2. Compilação do filtro com o Microsoft Visual C++ 6.0.

3. Realocação do `fmexample.exe` na pasta `[grail_dir]\bin`.

3.3.3.3 Criação de um Novo Módulo

A criação de um novo módulo requer alguns cuidados. Inicialmente, deve-se respeitar a estrutura padrão do Grail para Controle Supervisório. Em outras palavras, o módulo criado deve ser um pacote cujos diretórios estão descritos na Seção 3.3.2.

Além disso, suas classes devem ser derivadas das classes do Grail, sendo que, na maioria dos casos, essas serão derivadas da classe `FM`. A exemplo do Grail, cada classe do módulo terá um arquivo de descrição da classe `nome_da_classe.h`, a biblioteca de arquivos `include.h` e vários métodos, cujos algoritmos estejam presentes em arquivos do tipo `nome_do_metodo.src`.

Por último, o diretório `filters` deverá ser composto pelos `cpps` do filtros, pela biblioteca de classes `nome_do_modulo.h`, aonde estarão listados os `include.h` de todas as suas classes e também o `grail.h`[5], e pelos arquivos de entrada e saída, cujas funções deverão ser implementadas conforme a necessidade do programador.

Importante observar que o Grail segue um padrão em sua nomenclatura. Por exemplo, todos os filtros da classe `FM` começam com `FM`, todos os da classe `RE` começam com `RE` e todos os filtros de verificação de propriedades começam com `is`. O ideal é que este padrão de nomenclatura seja mantido na criação de novos módulos.

3.4 Conclusões

Este capítulo introduziu o ambiente Grail, assim como as mudanças que este sofreu antes e durante o desenvolvimento deste trabalho. O objetivo deste foi uma reestruturação do Grail na tentativa de deixá-lo mais modular e definir um procedimento de

[5]O módulo deverá ser implementado de forma a ser funcional quando descompactado dentro da pasta Grail, portanto o caminho que define a localização do `grail.h` é `../../filters`

expansão para o mesmo. O próximo capítulo apresenta os filtros do Grail para Controle Supervisório, ou seja, descreve as funcionalidades atualmente presentes neste ambiente.

Convém citar ainda que a descrição textual dos objetos de manipulação do Grail, assim como o seu uso extensivo de gabaritos, torna o ambiente em questão mais lento, fazendo-o perder, em termos de tempo de computação, para alguns outros programas de mesmo objetivo, como o TCT. Entretanto, o Grail é o que possui a expansão mais facilitada.

Capítulo 4

O Grail para Controle Supervisório

Neste capítulo são introduzidos os filtros presentes no Grail para Controle Supervisório Base, com destaque aos filtros implementados no decorrer deste trabalho. Pode-se dizer que este capítulo serve como um manual dos filtros do Grail para Controle Supervisório.

4.1 Filtros

4.1.1 Filtros do Grail Versão 2.5

Esta seção descreve os filtros presentes na versão 2.5 do Grail que permaneceram no Grail para Controle Supervisório, sendo estes referentes a operações sobre REs, a operações sobre FMs e a verificação de propriedades de FMs.

4.1.1.1 Expressões Regulares

1. `recat` - concatena duas REs[1].
   - **Sintaxe:** `recat re1 re2`
2. `restar` - faz o fechamento Kleene de uma RE[1].
   - **Sintaxe:** `restar re1`
3. `reunion` - faz a união entre duas REs[1].

[1]Quando as REs são capturadas pelo *prompt* do DOS, o símbolo 'Ì' é acrescido no final de cada expressão. Este problema não foi resolvido por não levar a erros maiores.

- **Sintaxe:** `reunion re1 re2`

4. `retofm` - computa uma FM que aceite a mesma linguagem que a RE dada como entrada. O resultado pode ser não determinista.

   - **Sintaxe:** `retofm re1`

5. `fmtore` - computa uma RE que aceite a mesma linguagem que a FM dada como entrada. A FM não precisa ser determinista e todos os estados não acessíveis são automaticamente removidos.

   - **Sintaxe:** `fmtore fm1`

4.1.1.2 Máquinas de Estados Finitos

1. `fmcat` - concatena duas FMs.

   - **Sintaxe:** `fmcat fm1 fm2`
   - **Descrição:** Conectam-se os estados marcados da `fm1` aos estados alcançáveis da `fm2`, por apenas uma transição, de algum dos estados iniciais de `fm2`.

2. `fmcment` - calcula o complemento de uma FM. O complemento de uma FM é dado por uma outra máquina de estados finitos que aceita qualquer palavra não aceita por FM, considerando-se como alfabeto de referência o alfabeto da própria FM. Um exemplo é ilustrado na Figura 4.1[2].

   - **Sintaxe:** `fmcment fm1`

3. `fmcomp` - completa uma FM. Uma FM completa é aquela em que todos os seus estados possuem transições etiquetadas com cada um dos eventos do seu alfabeto. Um exemplo é ilustrado na Figura 4.1.

   - **Sintaxe:** `fmcomp fm1`

4. `fmcross` - calcula o produto cartesiano de duas FMs, sendo que este cálculo gera a intersecção da linguagem das FMs em questão.

   - **Sintaxe:** `fmcross fm1 fm2`

[2]Importante observar que a maioria das FMs deste capítulo estão representadas graficamente para melhor visualização.

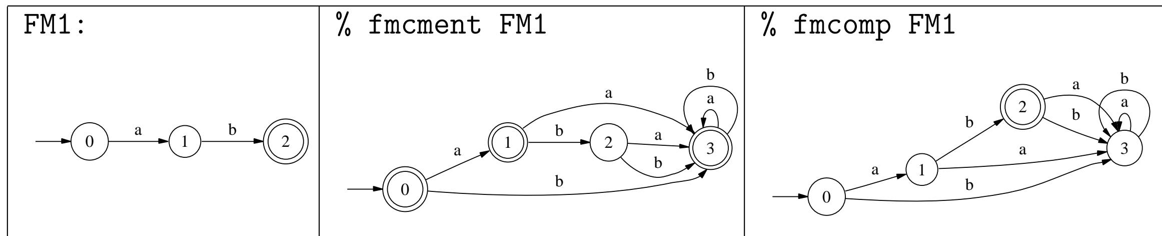


Figura 4.1: Exemplo - fmcment e fmcomp

A numeração dos estados da FM resultante segue a seguinte equação: $q = q_1 + n_1.q_2$, onde q é o número associado ao estado da FM resultante correspondente ao par (q_1, q_2), q_1 é um estado da `fm1`, q_2 é um estado da `fm2` e n_1 é o número de estados da `fm1`. Com base nesta equação, pode-se chegar a que estados cada estado da FM resultante é equivalente.

5. `fmdeterm` - torna uma FM determinista utilizando o método de construção de um equivalente determinista apresentado em [Cury, 2001].

   - **Sintaxe:** `fmdeterm fm1`

6. `fmenum` - enumera a linguagem marcada de uma FM. Produz 100 palavras, ou `num` palavras, caso o parâmetro `-n num` estiver especificado na chamada do filtro. As palavras são ordenadas de forma crescente e lexicográfica.

   - **Sintaxe:** `fmenum [-n num] fm1`

7. `fmexec` - executa uma FM para uma palavra de entrada. Em outras palavras, este filtro verifica se a palavra `str` pertence à linguagem gerada de `fm1`. A opção `-d` faz com que o `fmexec` escreva a lista de instruções que corresponde ao acompanhamento da palavra.

   - **Sintaxe:** `fmexec [-d] fm1 str`
   - **Contribuição:** Este filtro era funcional apenas para `fm<char>`. Como os filtros do Grail para Controle Supervisório utilizam como objetos `fm<string<char>>`, foi necessária uma reimplementação do mesmo. O algoritmo implementado no decorrer deste trabalho segue os seguintes passos:

     (a) Captura a FM de entrada e a palavra `str`, verificando se o usuário deseja acompanhar o andamento do algoritmo passo a passo - parâmetro `-d` presente.

     (b) Copia o conjunto de estados iniciais Q_i da FM para um conjunto de estados qualquer, denominado Q.

     (c) Verifica se o próximo elemento de `str`, iniciando a contagem a partir do primeiro elemento, é a etiqueta de transição de alguma instrução da FM tal que $\theta_{source} \in Q$.

– Em caso positivo, tem-se que se $(\theta_{source} \in Q) \wedge (\theta_{event} = str[i])$, então $\theta_{sink} \in Q'$. Faz-se $Q = Q'$ e retorna-se ao passo (c).

– Em caso negativo, a algoritmo é encerrado e `The string is NOT a member of the language of FM`.

(d) Se o algoritmo conseguir encontrar um novo conjunto de estados Q para todos os elementos de `str`, então `The string is a member of the language of FM`.

- **Exemplo:** A Figura 4.2 mostra a execução de duas palavras diferentes sobre uma determinada FM, sendo a primeira pertencente a linguagem da FM.

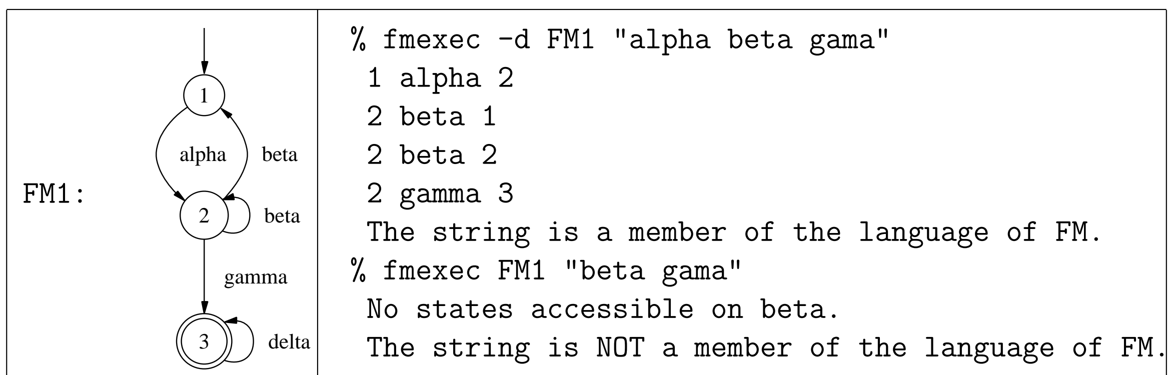


Figura 4.2: Exemplo - fmexec

8. `fmmin` - calcula o equivalente mínimo de uma FM utilizando uma extensão do algoritmo de Hopcroft. Não processa FMs não-deterministas. Um exemplo é apresentado na Figura 4.3.

   - **Sintaxe:** `fmmin fm1`

9. `fmminrev` - calcula o equivalente mínimo de uma FM pelo algoritmo de Brzozowski. Aceita FMs deterministas e também não deterministas. Um exemplo é apresentado na Figura 4.3.

   - **Sintaxe:** `fmminrev fm1`

   O Algoritmo de Brzozowski baseia-se na seguinte seqüência de operações:

   (a) Inverte o sentido das transições da FM;

   (b) Calcula o equivalente determinista;

   (c) Inverte novamente o sentido das transições;

   (d) Calcula o equivalente determinista novamente.

   Após a execução desta seqüência de operações, o resultado é o equivalente mínimo determinista da FM em questão, sendo este isomorfo ao resultado obtido felo filtro `fmmin`.

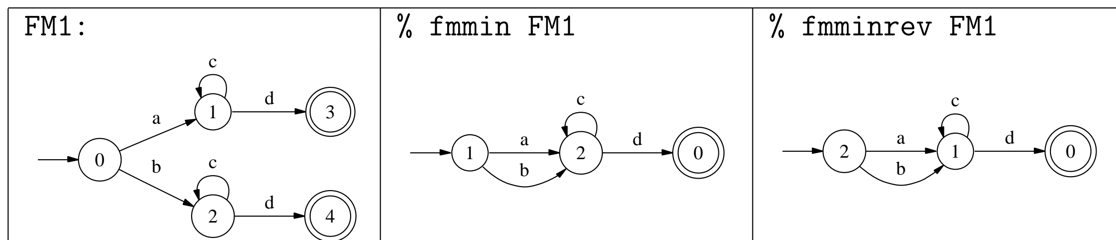


Figura 4.3: Exemplo - fmmin e fmminrev

10. `fmplus` - calcula o `plus` de uma FM, isto é, computa uma FM que reconhece uma ou mais ocorrências de palavras reconhecidas pela FM de entrada.
    - **Sintaxe:** `fmplus fm1`
11. `fmreach` - computa a componente acessível de uma FM.
    - **Sintaxe:** `fmreach fm1`
12. `fmrenum` - renumera os estados de uma FM de acordo com um esquema de numeração canônico, crescente em relação aos estados fontes das instruções e lexicográfico sobre as etiquetas de transição. Não processa FMs não deterministas.
    - **Sintaxe:** `fmrenum fm1`
13. `fmreverse` - inverte o sentido de todas as transições da FM, tornando finais os estados iniciais e vice-versa.
    - **Sintaxe:** `fmreverse fm1`
14. `fmstar` - calcula o Fechamento Kleene da FM.
    - **Sintaxe:** `fmstar fm1`
15. `fmstats` - obtém informações sobre a FM, tais como número de estados, transições e símbolos, assim como número de estados iniciais, finais, acessíveis, coacessíveis e bloqueantes. Um exemplo é ilustrado na figura 4.4.
    - **Sintaxe:** `fmstats fm1`
16. `fmunion` - calcula a união de duas FMs renumerando os estados da segunda FM. O resultado é não-determinista.
    - **Sintaxe:** `fmunion fm1 fm2`

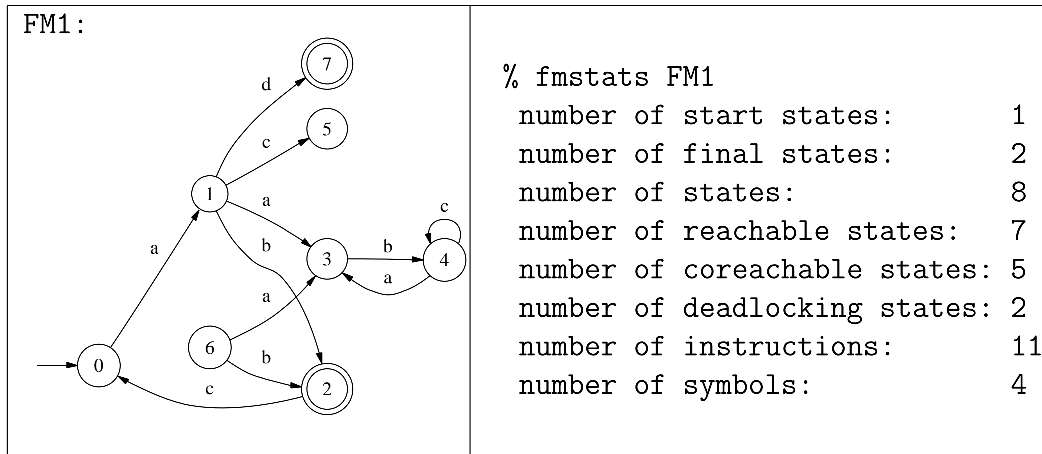


Figura 4.4: Exemplo - fmstats

4.1.1.3 Verificação de Propriedades

1. `iscomp` - verifica se uma FM é completa. Lembrando que uma FM é completa se todos os seus estados possuem uma transição para cada evento do seu alfabeto.
    - **Sintaxe:** `iscomp fm1`
    - **Saída:** Escreve `'FM is complete'` na saída padrão se `fm1` for completa e `'FM is NOT complete'`, caso contrário.
2. `isdeterm` - verifica se uma FM é determinista.
    - **Sintaxe:** `isdeterm fm1`
    - **Saída:** Escreve `'FM is deterministic'` na saída padrão se `fm1` for determinista e `'FM is NOT deterministic'`, caso contrário.
3. `isomorph` - verifica se duas FMs deterministas são isomorfas.
    - **Sintaxe:** `isomorph fm1 fm2`
    - **Saída:** Escreve `'The FMs are isomorphic'` na saída padrão se `fm1` e `fm2` forem iguais a menos da numeração dos estados e `'The FMs are NOT isomorphic'`, caso contrário.
4. `isuniv` - verifica se uma FM é universal, isto é, se ela é completa e se todos os seus estados acessíveis são também finais.
    - **Sintaxe:** `isuniv fm1`
    - **Saída:** Escreve `'FM is universal'` na saída padrão se `fm1` for universal e `'FM is NOT universal'`, caso contrário.

4.1.2 Filtros Referentes ao Controle Supervisório

Esta seção explicita os filtros desenvolvidos para o Controle Supervisório, tendo estes sido implementados por José Miguel Eyzell González ou por Antonio Eduardo Carrilho da Cunha.

4.1.2.1 Máquinas de Estados Finitos

1. `fmalpha` - obtém o alfabeto Σ de uma FM.
   - **Sintaxe:** `fmalpha fm1`
2. `fmloop` - faz o *selfloop* de eventos numa FM.
   - **Sintaxe:** `fmloop fm1 fm2`

   Este filtro cria, para cada estado de `fm1`, transições do estado para ele mesmo, *selfloops*, envolvendo todos os eventos pertencentes ao alfabeto da `fm2`.
3. `fmmark` - marca todos os estados de uma FM, ou seja, faz com que todos os estados da FM sejam finais.
   - **Sintaxe:** `fmmark fm1`
4. `fmproj` - faz a projeção de uma FM sobre o alfabeto de uma segunda FM. Um exemplo é ilustrado na Figura 4.5
   - **Sintaxe:** `fmproj fm1 fm2`

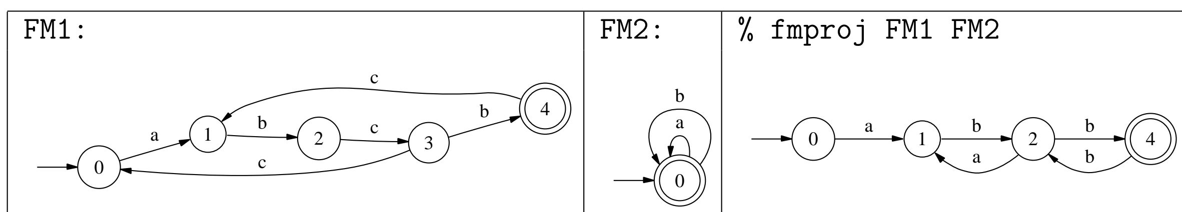


Figura 4.5: Exemplo - fmproj

5. `fmremove` - elimina os estados de uma FM.
   - **Sintaxe:** `fmremove fm1 fm2`

     Remove-se os estados de `fm1`, cujos números foram passados como eventos em `fm2`, e suas respectivas transições. Um exemplo é ilustrado na Figura 4.6.

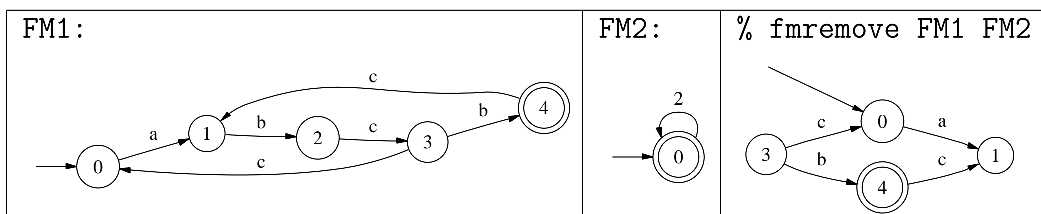


Figura 4.6: Exemplo - fmremove

6. `fmsort` - ordena as transições de uma FM, primeiro pelo número dos estados fontes das transições, depois pela ordem lexicográfica dos eventos e, por último, pelo número dos estados destino das transições.

   - **Sintaxe:** `fmsort fm1`

7. `fmsync` - calcula a composição síncrona de duas FMs. A numeração dos estados segue a mesma equação apresentada para o filtro `fmcross`, Seção 4.1.1.2.

   - **Sintaxe:** `fmsync fm1 fm2`
   - **Contribuição:** Sejam as FMs P_1, P_2 e P_3, mostradas na Figura 4.7, subsistemas que compõem uma planta P. O evento `alpha`, presente nos três subsistemas, é eliminado no produto síncrono de P_1 com P_2. Conseqüentemente, o resultado final, a planta P, computado pelo seguinte comando `% fmsync P1 P2 | fmsync P3`, ou `% fmsync P2 P1 | fmsync P3`, não é correto. Isto acontece porque o Grail considera que a máquina $P_{12} = P_1 \| P_2$ não possui o evento `alpha`, tornando ele habilitado em todos os estados de P_{12} para a composição com P_3. Mas, neste caso, `alpha` está desabilitado em todos os estados de P_{12}.

     Esse erro é gerado sempre que a composição síncrona de várias máquinas é computada seqüencialmente e que o produto de duas máquinas desta seqüência elimina algum evento presente também numa terceira máquina, a ser composta posteriormente. Este não é um erro inerente ao Grail, sendo que outros programas, como por exemplo o TCT, também o possuem.

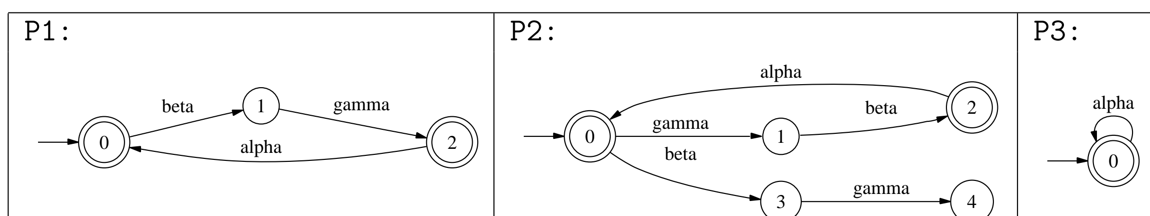


Figura 4.7: Problema - fmsync

Um aviso de alerta foi adicionado ao filtro `fmsync`, sendo este ativado sempre que o resultado da composição de duas FMs não possuir algum evento presente nestas FMs. A resolução do exemplo da Figura 4.7 é apresentada abaixo.

```
% fmsync P1 P2 | fmsync P3 > P
 WARNING! The synchronous composition of P1 and P2 does not
 have the event(s) 'alpha'.
```

Como o problema não ocorreu na última composição, e sim na de P_1 com P_2, e como P_3 possui o evento `alpha`, o resultado obtido está incorreto. O usuário, ao se deparar com um aviso de alerta como este, deve inverter a ordem da composição das máquinas.

Caso a inversão na ordem dos parâmetros não seja suficiente para eliminar o problema, basta adicionar ao resultado obtido a partir do produto síncrono que gera o erro um estado inacessível que tenha um *selfloop* com os eventos que estão sendo eliminados.

8. `fmtrim` - encontra a componente `trim` de uma FM.
   - **Sintaxe:** `fmtrim fm1`
9. `fmcondat` - encontra os eventos de uma FM desabilitados por uma segunda FM. São dois os parâmetros deste filtro, sendo `fm1` a planta e `fm2` a FM que modela a linguagem-alvo.
   - **Sintaxe:** `fmcondat fm1 fm2`
10. `fmsupc` - computa a máxima linguagem controlável. São três os parâmetros deste filtro, sendo que `fm1` representa a planta, `fm2` a FM que modela a linguagem-alvo e `fm3` o conjunto de eventos não-controláveis. Convém observar que os eventos não-controláveis são passados como o alfabeto de `fm3`.
    - **Sintaxe:** `fmsupc fm1 fm2 fm3`
11. `fmtodot` - converte uma FM para o formato reconhecido pela ferramenta gráfica Graphviz.
    - **Sintaxe:** `fmtodot fm1 > fm1.do`
12. `fmtovcg` - converte uma FM para o formato reconhecido pela ferramenta VCG.
    - **Sintaxe:** `fmtovcg fm1 > fm1.vcg`

4.1.2.2 Verificação de Propriedades

1. `iscreach` - verifica se uma FM é coacessível.
   - **Sintaxe:** `iscreach fm1`
   - **Saída:** Escreve `'FM is coreachable'` na saída padrão se algum estado final é alcançável a partir de todos os estados de `fm1`.

2. `islock` - verifica se uma FM é não-bloqueante. A noção de bloqueio associada ao Grail para Controle Supervisório está relacionada com a idéia de um autômato que possui estados a partir dos quais só a cadeia vazia ϵ é aceita, sendo que estados finais também podem ser bloqueantes.
   - **Sintaxe:** `islock fm1`
   - **Saída:** Escreve `'FM does NOT have any deadlocking state'` na saída padrão se `fm1` não possuir estados a partir dos quais nenhuma transição sai.
3. `isreach` - verifica se uma FM é acessível.
   - **Sintaxe:** `isreach fm1`
   - **Saída:** Escreve `'FM is reachable'` na saída padrão se todos os estados de `fm1` forem alcançáveis a partir do estado inicial.

A Figura 4.8 mostra a saída dos filtros `iscreach`, `islock` e `isreach` para uma FM que não é coacessível nem acessível, porém é bloqueante.

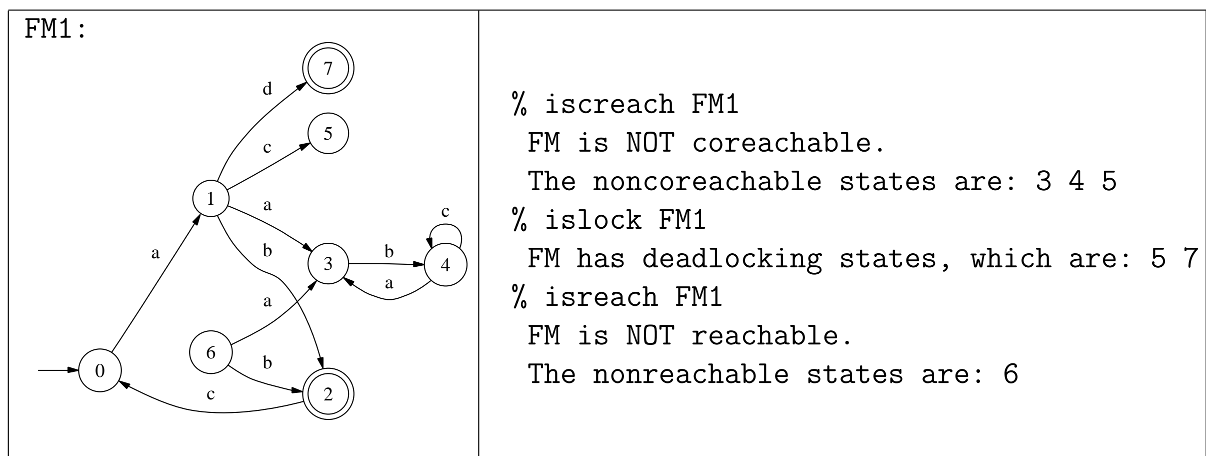


Figura 4.8: Exemplo - iscreach, islock e isreach

4.1.3 Novos Filtros

Esta seção explicita os filtros para o Controle Supervisório desenvolvidos no decorrer deste trabalho.

4.1.3.1 Verificação do L-Fechamento

Como já citado anteriormente, uma linguagem-alvo K é $L_m(P)$-fechada se $K = \overline{K} \cap L_m(P)$ e esta é uma das propriedades que garantem a existência de um supervisor. Por

isso, tornou-se interessante a existência de um filtro do Grail para Controle Supervisório que verificasse esta propriedade.

A sintaxe deste filtro é dada por `% islclosed fm1 fm2`, sendo `fm1` a planta e `fm2` a FM que modela a linguagem-alvo K. O algoritmo que implementa este filtro consiste nos seguintes passos:

1. Captura dos parâmetros de entrada `fm1` e `fm2`.
2. Cálculo de uma FM, denominada `temp`, que represente a linguagem dada por $\overline{K} \cap L_m(P)$. Sintaxicamente: `% fmmark fm2 | fmcross fm1 > temp`.
3. Verificação do isomorfismo entre `temp` e `fm1`.
   - Em caso positivo, `The desired language is L-closed wrt the plant`.
   - Em caso negativo, `The desired language is NOT L-closed wrt the plant`.

4.1.3.2 Verificação da Controlabilidade

A controlabilidade da linguagem-alvo K em relação à planta P é também uma das propriedades que garantem a existência de um supervisor. Uma linguagem $K \subseteq \Sigma^*$ é controlável em relação a P se $\overline{K}\Sigma_u \cap L(P) \subseteq \overline{K}$.

A sintaxe deste filtro é dada por `% iscontrol fm1 fm2 fm3`, sendo que `fm1` representa a planta, `fm2` a FM que modela K e `fm3` o conjunto de eventos não-controláveis. O algoritmo que implementa este filtro é baseado no algoritmo do `fmcondat` e consiste nos seguintes passos:

1. Captura dos parâmetros de entrada `fm1`, `fm2` e `fm3`.
2. Cálculo da componente acessível de uma FM que representa a intersecção entre a linguagem da planta e a linguagem-alvo, denominada `temp`.
3. Para cada estado de `temp`, temos que:
   (a) Capturar os eventos que estão habilitados em `fm1` e desabilitados em `temp`. Estes caracterizam os eventos da planta a serem desabilitados pela especificação.
   (b) Comparação desses com os eventos não-controláveis, obtidos a partir da `fm3`.
      - Se existem eventos não-controláveis desabilitados, o algoritmo é encerrado e `The desired language is NOT controllable wrt the plant`.

- Se não existem eventos não-controláveis desabilitados, o algoritmo retorna ao passo 3.

4. Encerramento do algoritmo, retornando que `The desired language is controllable wrt the plant`.

4.1.3.3 Verificação do Não-Conflito

A propriedade do não-conflito, a ser verificada entre vários supervisores modulares locais S_j é verificada se e somente se $S = \|_{j=1}^{m} S_j$ for um autômato *trim*, como mostra a Seção 2.3.2.

Com base nesta afirmação, o filtro `isnconf` em sua sintaxe dada por `% isnconf fm1 fm2 ... fmn` e uma implementação que consiste nos seguintes passos:

1. Captura de todos supervisores modulares locais;
2. Computação do produto síncrono de todos os supervisores passados como parâmetro;
3. Verificação da acessibilidade e coacessibilidade do autômato resultante.
   - Se o autômato resultante for *trim*, então `The FMs are conflicting`.
   - Se o autômato resultante não for *trim*, então `The FMs are NOT conflicting`.

4.1.3.4 Redutor de Supervisores

Como já citado anteriormente na Seção 2.3.3, a redução de supervisores é feita por intermédio da agregação de estados compatíveis entre si em blocos, não necessariamente disjuntos entre si.

Segundo [Su and Wonham, 2003], sendo $S = (Q, \Sigma, \delta, q_0, Q_m)$ um supervisor da planta $P = (Q_P, \Sigma, \delta_P, q_{P,0}, Q_{P,m})$, define-se :

- $E : Q \rightarrow 2^{\Sigma}$ como sendo o conjunto de eventos habilitados por S, tal que $E(q) = \{\sigma \in \Sigma | \delta(q, \sigma)!\}$;
- $D : Q \rightarrow 2^{\Sigma}$ como sendo o conjunto de eventos desabilitados por S, tal que $D(q) = \{\sigma \in \Sigma | \neg\delta(q, \sigma)! \wedge (\exists s \in \Sigma^*)[\delta(q_0, s) = q \wedge \delta_P(q_{P,0}, s\sigma)!]\}$;
- $M : Q \rightarrow \{true, false\}$ como sendo o conjunto dos estados marcados do supervisor, definido como $M(q) = true$ caso $q \in Q_m$;

- $T : Q \rightarrow \{true, false\}$ como sendo o conjunto dos estados marcados da planta alcançáveis sob supervisão, definido como $T(q) = true$ caso $(\exists s \in \Sigma^*)[\delta(q_0, s) = q \wedge \delta_P(q_{P,0}, s) \in Q_{P,m}]$.

Uma cobertura $C = \{B_i \subseteq Q | i \in I\}$ é uma *cobertura de controle* de S se:

1. $(\forall i \in I) B_i \neq \emptyset$;
2. $(\forall q, q' \in B_i)[E(q) \cap D(q') = E(q') \cap D(q) = \emptyset]$, ou seja, as ações de controle de q e q' devem ser consistentes;
3. $(\forall q, q' \in B_i)[T(q) = T(q') = true \Rightarrow M(q) = M(q')]$, o que implica que estados marcados na planta e no supervisor, denominados marcadores, são incompatíveis com os estados chamados de desmarcadores, por estarem marcados na planta e desmarcados no supervisor;
4. $(\forall i \in I)(\forall \sigma \in \Sigma)(\exists j \in I)[(\forall q \in B_i)\delta(q, \sigma)! \Rightarrow \delta(q, \sigma) \in B_j]$, ou seja, o conjunto de estados que pode ser alcançado a partir de algum estado de B_i através de apenas um evento é coberto por algum $B_j \in C$.

Em palavras, cobrir S consiste basicamente numa divisão do seu conjunto de estados Q em subconjuntos de estados não-vazios B_i, denominados blocos, tal que estados pertencentes a um mesmo bloco se comportam consistentemente sob a ação da função de transição δ e exibem uma ação de controle uniforme nos estados em que esse importa e tal que estados marcadores e desmarcadores não podem pertencer a um mesmo bloco.

Dada uma cobertura de controle $C = \{B_i \subseteq Q | i \in I\}$ de S, podemos construir um *supervisor induzido* $I = (Q_I, \Sigma, \delta_I, Q_{I,0}, Q_{I,m})$ tal que:

1. $i_0 =$ algum $i \in I$ tal que $q_0 \in B_i$;
2. $I_m = \{i \in I | B_i \cap Q_m \neq \emptyset\}$;
3. $\xi : I \times \Sigma \rightarrow I$ com $\xi(i, \sigma) = j$ provido pela escolha de um $j \in I$ tal que se $(\exists q \in B_i)\delta(q, \sigma) \in B_j$, então $(\forall q' \in B_i)[\delta(q', \sigma)! \Rightarrow \delta(q', \sigma) \in B_j]$.

Sendo S um supervisor de P e S_{red} um supervisor equivalente em controle à S com relação à P, então existe uma cobertura de controle C de S para a qual algum supervisor induzido I seja isomorfo a S_{red} [Su and Wonham, 2003].

O algoritmo que implementa o redutor de supervisores do Grail para Controle Supervisório, `fmsupred`, consiste na computação de uma cobertura de controle e na posterior criação de um supervisor induzido com base nesta cobertura. A sintaxe

deste filtro é dada por `% fmsupred fm1 fm2`, sendo que `fm1` modela a planta e `fm2` o supervisor.

O `fmsupred` inicia sua computação a partir de uma cobertura de controle que possui apenas um bloco, cujo único estado pertencente é o estado inicial. Os estados alcançáveis a partir do estado inicial com apenas um evento são, então, utilizados como base para a expansão do bloco em questão ou a criação de um novo bloco. O algoritmo entra num *loop* até que todos os estados acessíveis estejam todos presentes em pelo menos um bloco.

Em resumo, o redutor de supervisores do Grail para Controle Supervisório obtém o supervisor induzido I através da computação de uma cobertura de controle, cujos blocos não são necessariamente disjuntos como no algoritmo proposto por [Su and Wonham, 2003]. Entretanto, esta cobertura é determinada de forma induzida, o que não garante a obtenção da cobertura com o menor número de estados possível - supervisor mínimo. Essa premissa é válida porque a computação do supervisor mínimo é NP-Difícil [Vaz and Wonham, 1986], enquanto que o algoritmo em questão possui complexidade polinomial.

4.1.3.5 Conversão de Formatos

Além do Grail, existem outras ferramentas computacionais que lidam com a Teoria de Controle Supervisório. Com o objetivo de tornar o Grail compatível com algumas destas outras ferramentas, foram implementados no decorrer deste trabalho filtros de conversão de formatos que abrangem ferramentas como o TCT, CTCT e STCT [Wonham, 2005]. São eles:

- **fmtopds:** converte uma FM para o formato aceito pelo TCT. Convém citar que o TCT identifica os eventos ímpares como controláveis e os pares como não-controláveis. Por isso, a presença de um segundo parâmetro é optativa. Quando presente, este parâmetro representa o conjunto de eventos não controláveis. Quando ausente, todos os eventos são considerados controláveis. Um exemplo é apresentado na Figura 4.9.
  - **Sintaxe:** `fmtopds fm1 fm2 > fm1.pds`
- **fmtoads:** converte uma FM para o formato aceito pelo CTCT. A exemplo do filtro anterior, o segundo parâmentro é optativo e representa os eventos não controláveis. Um exemplo é apresentado na Figura 4.10.
  - **Sintaxe:** `fmtoads fm1 fm2 > fm1.ads`

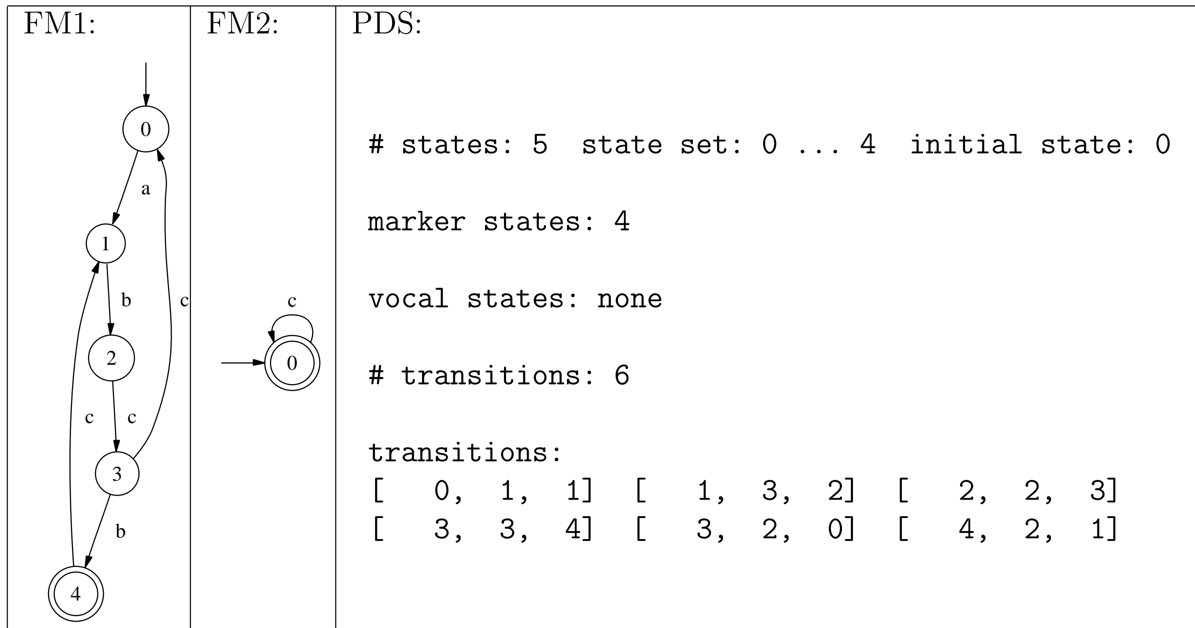


Figura 4.9: Exemplo - fmtopds

- **fmtofsm:** converte uma FM para o formato aceito pelo STCT. A exemplo dos filtros anteriores, o segundo parâmentro é optativo e representa os eventos não controláveis. Um exemplo é apresentado na Figura 4.11.
  - **Sintaxe:** `fmtofsm fm1 fm2 > fm1.fsm`

4.2 Exemplo - Mesa Giratória

Para que seja possível a resolução do problema da Mesa Giratória com o auxílio do ambiente Grail para Controle Supervisório, é necessária a inclusão dos subsistemas da planta e das restrições em arquivos textos, cujo modelo foi apresentado no Capítulo 3.

Inicialmente, cada dispositivo da célula de manufatura, M_0, M_1, M_2, M_3 e M_4, é armazenado num arquivo ASCII. Como exemplo, apresenta-se, na Figura 4.12, o arquivo ASCII referente ao modelo da máquina M_0 e o seu autômato equivalente. Todas as restrições são também representadas segundo o mesmo modelo, a exemplo da restrição E_a apresentada na Figura 2.7.

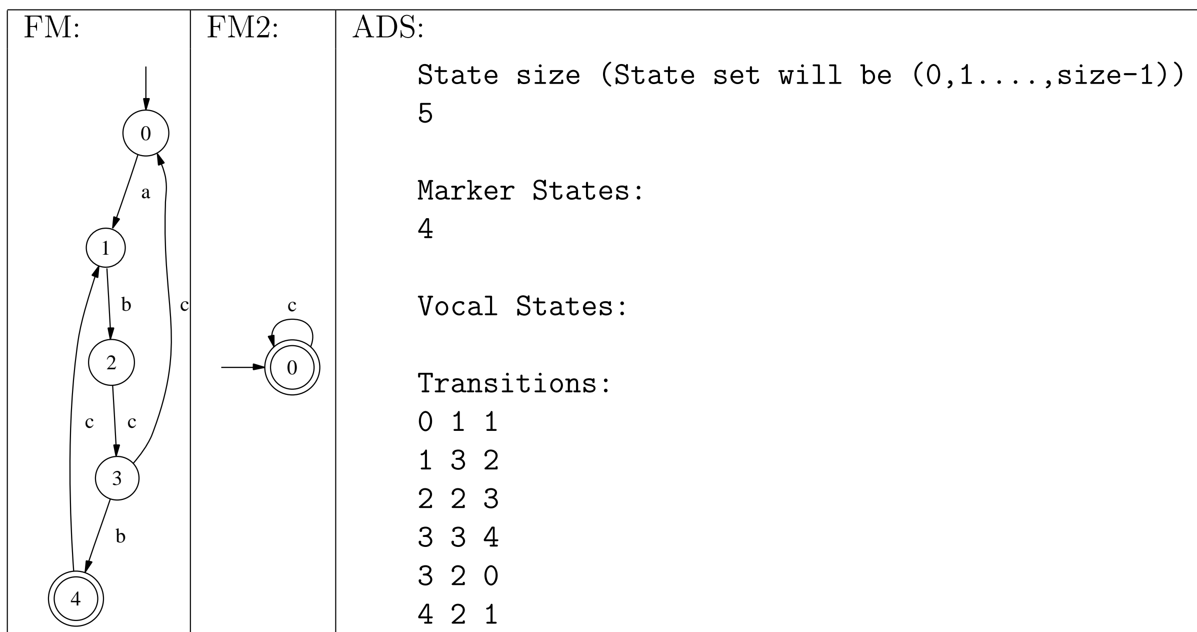


Figura 4.10: Exemplo - fmtoads

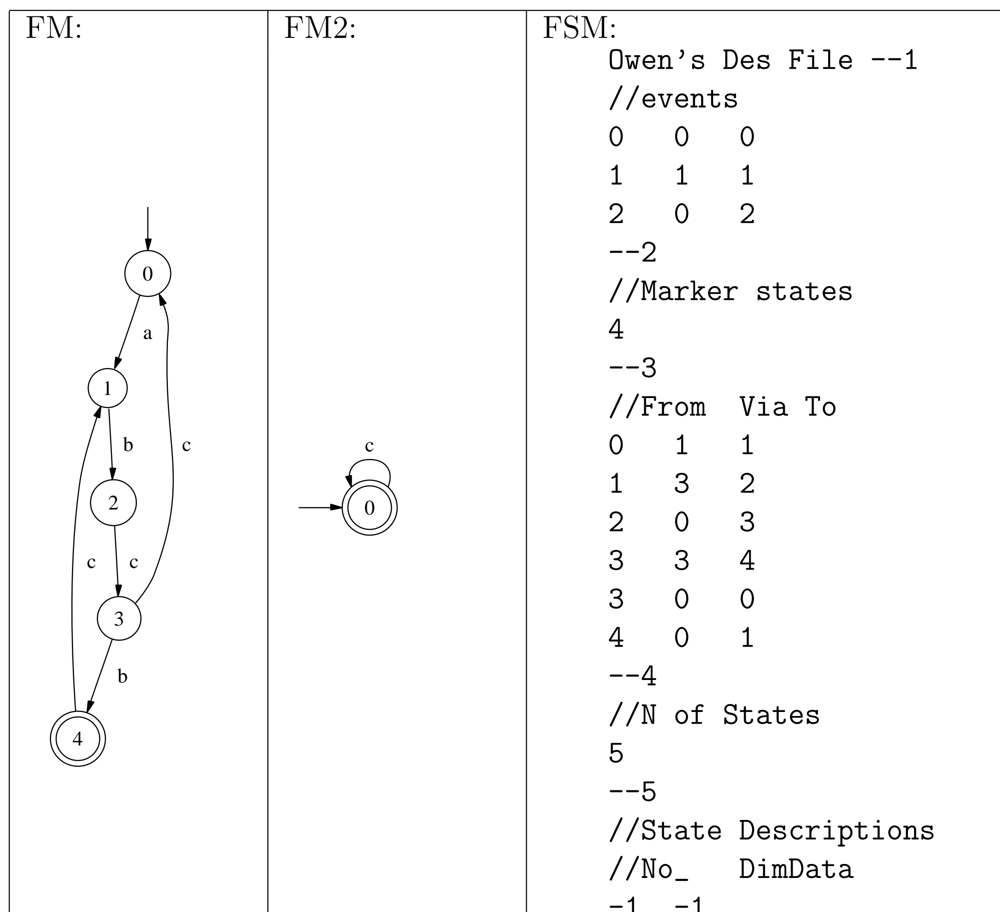


Figura 4.11: Exemplo - fmtofsm

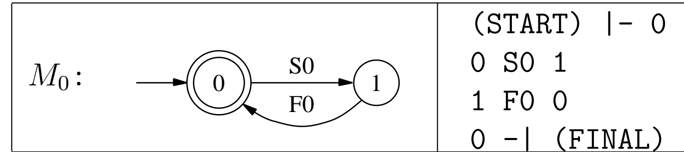
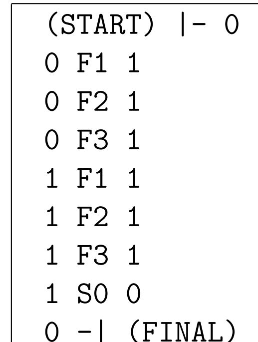


Figura 4.12: Representação no Grail do Dispositivo M_0



```
(START) |- 0
0 F1 1
0 F2 1
0 F3 1
1 F1 1
1 F2 1
1 F3 1
1 S0 0
0 -| (FINAL)
```

Figura 4.13: Representação no Grail da Restrição E_a

A parte da modelagem está concluída com cada subsistema da planta e cada restrição de coordenação armazenado num arquivo. Pode-se, agora, aplicar as funções do Grail para Controle Supervisório com o objetivo de encontrar o supervisor ótimo. Como já vimos anteriormente, a resolução deste problema pode ser feita através de diferentes abordagens, dentre elas a monolítica e a modular local.

4.2.1 Síntese do Supervisor Monolítico

A resolução deste problema de acordo com a abordagem monolítica se dá através dos seguintes passos.

1. Construção da planta livre:

```
% fmsync m0 m1 | fmsync m2 | fmsync m3 | fmsync m4 > P
```

2. Criação de um arquivo com os eventos não-controláveis da planta:

```
% type ncont
 0 F0 0
 0 F1 0
 0 F2 0
 0 F3 0
 0 F4 0
```

3. Construção da especificação E:

```
% fmsync Eb1 Eb2 | fmsync Eb3 | fmsync Eb4 > Eb
% fmsync Ec1 Ec2 | fmsync Ec3 > Ec
% fmsync Ea Eb | fmsync Ec > E
```

4. Obtenção do autômato que representa a linguagem-alvo:

```
% fmsync P E > K
% fmtrim K > Ktrim
```

5. Determinação do supervisor minimamente restritivo e obtenção de algumas informações sobre o mesmo:

```
% fmsupc P Ktrim ncont > Super
% fmstats Super
 number of start states:        1
 number of final states:        1
 number of states:              151
 number of reachable states:    151
 number of coreachable states:  151
 number of nonblocking states:  0
 number os instructions:        350
 number of symbols:             10
```

6. Redução do supervisor minimamente restritivo e observação de algumas informações sobre o resultado obtido:

```
% fmsupred P Super > SuperRed
% fmstats SuperRed
 number of start states:        1
 number of final states:        1
 number of states:              55
 number of reachable states:    55
 number of coreachable states:  55
 number of nonblocking states:  0
 number os transitions:         174
 number of symbols:             10
```

Mesmo após a redução, o supervisor obtido ainda é de difícil legibilidade e, conseqüentemente, de difícil implementação. Isso sem levar em consideração o tempo de computação referente à redução do supervisor, no caso 42 segundos[3], visto que o algoritmo de redução é polinomial ao número de estados do supervisor.

[3]O computador utilizado é um AMD Athlon(tm) XP 2700+ 2,16GHz e 512MB de RAM.

4.2.2 Síntese dos Supervisores Modulares Locais

A resolução deste problema de acordo com a abordagem modular local envolve os mesmos passos que a abordagem monolítica. A diferença está apenas na construção da planta livre, que neste caso é local, e na ausência do passo 3.

A síntese do supervisor modular local para a restrição E_{c1} é feita como descrito a seguir.

1. Construção da planta local livre:

```
% fmsync m0 m1 | fmsync m2 > Pc1
```

2. Criação de um arquivo com os eventos não-controláveis da planta:

```
% type ncontc1
 0 F0 0
 0 F1 0
 0 F2 0
```

3. Obtenção do autômato que representa a linguagem-alvo:

```
% fmsync Pc1 Ec1 | fmtrim > Kc1
```

4. Determinação do supervisor minimamente restritivo e obtenção de algumas informações sobre o mesmo:

```
% fmsupc Pc1 Kc1 ncontc1 > Sc1
% fmstats Sc1
 number of start states:        1
 number of final states:        1
 number of states:              24
 number of reachable states:    24
 number of coreachable states:  24
 number of nonblocking states:  0
 number os transitions:         52
 number of symbols:             6
```

5. Redução do supervisor minimamente restritivo e observação de algumas informações sobre o resultado obtido:

```
% fmsupred Pc1 Sc1 > SRedc1
% fmstats SReda
 number of start states:        1
 number of final states:        1
 number of states:              8
 number of reachable states:    8
 number of coreachable states:  8
 number of nonblocking states:  0
 number of transitions:         26
 number of symbols:             6
```

O tempo de computação do supervisor reduzido $S_{red,c1}$, Figura 4.14, foi de menos de 1 segundo[3].

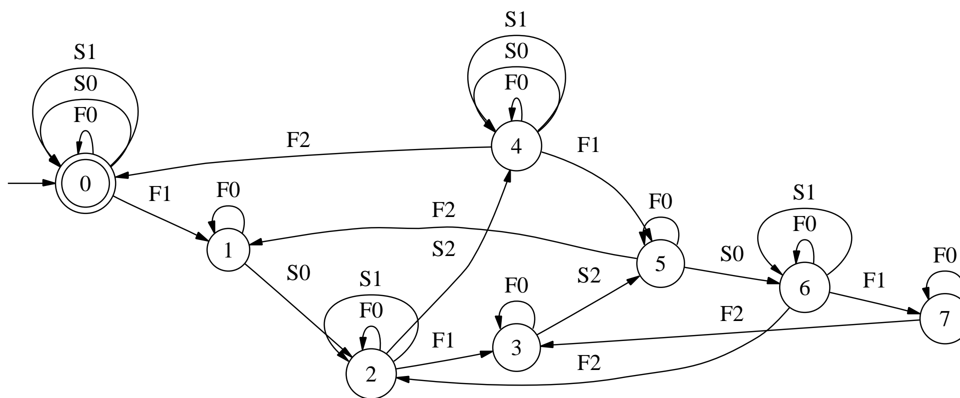


Figura 4.14: Supervisor Modular Local para E_{c1} reduzido

Os passos 1 a 5 devem ser repetidos para cada uma das restrições E_a, E_{b1}, E_{b2}, E_{b3}, E_{b4}, E_{c2} e E_{c3}, obtendo-se, desta forma, o supervisor modular local referente a cada uma destas restrições.

Um último passo a ser realizado é o teste de modularidade, que pode ser realizado da seguinte forma:

```
% isnconf SReda SRedb1 SRedb2 SRedb3 SRedb4 SRedc1 SRedc2 SRedc3
 The FMs are NOT conflicting.
```

Se supervisores locais forem modulares, como neste caso, o resultado obtido é equivalente ao obtido através da abordagem monolítica.

4.3 Conclusões

Este capítulo teve como objetivo a familiarização do leitor com os filtros atualmente disponíveis no Grail para Controle Supervisório. No decorrer deste trabalho, alguns filtros para o Controle Supervisório, com destaque ao redutor de supervisores, foram implementados. Além disso, todos os filtros receberam um texto de ajuda - *help*, acionado com o parâmetro `-h` ou com o recebimento de um número errado de parâmetros.

Filtros que lidam com extensões da Teoria de Controle Supervisório, tais como Sistemas Condição/Evento, Sistemas a Eventos Discretos com Marcação Flexível e Controle Multitarefa também foram desenvolvidos no decorrer deste trabalho. Estes filtros foram acrescidos ao Grail para Controle Supervisório como módulos, descritos nos próximos capítulos.

Capítulo 5

Controle Multitarefa

Em muitos problemas reais, diversos tipos diferentes de tarefas são executados. Nestas situações, a modelagem da planta por um único autômato pode ser problemática, já que a realização de qualquer tipo de tarefa seria identificada pela mesma marcação. Este capítulo apresenta, baseado em [de Queiroz, 2004], uma abordagem para o tratamento de múltiplos tipos de tarefas no controle supervisório de SEDs e, também, o módulo do Grail que lida com esta abordagem.

Quando um SED inclui múltiplos tipos de tarefas, é chamado de Sistemas a Eventos Discretos Multitarefa (SEDMT). Para a modelagem de SEDMTs, introduz-se conceito de comportamento colorido, Seção 5.1, e também uma forma direta de representação deste, o Autômato de Marcação Colorida (AMC), Seção 5.2. A partir daí, os principais resultados da Teoria de Controle Supervisório são estendidos para este novo tipo de autômato, Seção 5.3, o que permite a síntese de supervisores mais refinados em problemas de controle nos quais a distinção de tipos de tarefas é necessária. A Seção 5.4 descreve o módulo Multitarefa, totalmente desenvolvido no decorrer deste trabalho. Por último, na Seção 5.5, um exemplo ilustra a conveniência da abordagem em questão e a utilização dos filtros aqui implementados.

5.1 Comportamento Colorido

Com o intuito de diferenciar os múltiplos tipos de tarefas de um SEDMT, são associadas cores a estes. Seja Σ o conjunto de todos os eventos que podem ocorrer no sistema e C o conjunto de todas as cores. Para cada cor $c \in C$ pode-se associar uma linguagem $L_c \in 2^{\Sigma^*}$ que representa o conjunto de todas as seqüências de eventos de Σ que completam tarefas do respectivo tipo. Desse modo, define-se o comportamento colorido $\Lambda_C \in 2^{2^{\Sigma^*} \times C}$ de um SEDMT como o conjunto de pares $\{(L_c, c), c \in C\}$.

Para um comportamento colorido Λ_C, a linguagem marcada por $\emptyset \subset B \subseteq C$ é definida por $L_B(\Lambda_C) = \cup_{b \in B}\{L : (L, b) \in \Lambda_C\}$. A linguagem gerada por Λ_C é o conjunto de todas as cadeias que podem completar qualquer tarefa de C, isto é, $L(\Lambda_C) = \overline{L_C(\Lambda_C)}$.

Em alguns problemas, pode acontecer que um SED não inclua nenhum tipo de tarefa e, portanto, o conjunto de cores seja vazio. Para esse caso particular, o comportamento pode ser descrito simplesmente por uma linguagem prefixo-fechada L contendo qualquer seqüência de eventos que possa acontecer no sistema. Para representar tal comportamento como um comportamento colorido, pode-se reservar a cor `v` (de *void*) para uma tarefa sem significado que é completada por qualquer cadeia de L. Então, o comportamento colorido de tal SED seria dado por $\Lambda_C = \{(L, v),\ com\ C = \{v\}\}$.

5.2 Autômato de Marcação Colorida

Um SEDMT pode ser modelado por um autômato especial, cujos estados são marcados por subconjuntos de cores de acordo com os tipos de tarefas completadas. Essa particular máquina de estados, denominada Autômato de Marcação Colorida (AMC), é definida formalmente como uma sêxtupla $A = (Q, \Sigma, C, \delta, \chi, q_0)$, onde:

- Q representa um conjunto não-vazio de estados;
- Σ é o conjunto de eventos que definem o alfabeto;
- C representa o conjunto de cores;
- $\delta : Q \times \Sigma \rightarrow Q$ é a função de transição de estados;
- $\chi : Q \rightarrow 2^C$ representa a função de marcação;
- $q_0 \in Q$ é o estado inicial.

Pode-se associar ao AMC A uma função de eventos ativos $\Gamma : Q \rightarrow 2^\Sigma$, que associa cada estado $q \in Q$ a um subconjunto de Σ com todos os eventos que possam ocorrer em q, ou seja, $\Gamma(q) = \{\sigma \in \Sigma : \delta(q, \sigma)!\}$.

Esse modelo acrescenta ao autômato usual um conjunto de cores C, representando todos os tipos de tarefas que um sistema pode executar, e uma função de marcação χ, que atribui a cada estado de Q um subconjunto de cores. Conseqüentemente, um AMC é, basicamente, um autômato de Moore [Moore, 1964], cujas saídas, representadas por subconjuntos de cores, definem os tipos de tarefas que são completadas após uma seqüência de eventos.

5.2.1 Linguagens Associadas a um AMC

A linguagem gerada por um AMC A, denotada por $L(A)$, representa todas as possíveis cadeias finitas de eventos que são alcançadas a partir do estado inicial q_0. Como esse conceito independe da marcação, $L(A)$ é formalmente definida da mesma forma que a linguagem gerada por um autômato usual ($L(A) = \{s \in \Sigma^* : \delta(q_0, s)!\}$).

A linguagem marcada de um autômato representa o conjunto de cadeias de eventos gerada que completam uma tarefa. Como um AMC costuma envolver múltiplos tipos de tarefas, pode-se definir uma linguagem marcada para cada tipo como o conjunto de cadeias que levam a estados cujas funções de marcação contenham a cor relativa àquela classe. Assim, $L_c(A)$, a linguagem marcada por $c \in C$, é formalmente definida por: $L_c(A) = \{s \in L(A) : c \in \chi(\delta(q_0, s))\}$.

O conceito de linguagem marcada por uma cor pode ser estendido para um subconjunto não-vazio de cores como o conjunto de cadeias de eventos que completam qualquer tarefa representada por alguma das cores em questão. Então, para um conjunto de cores $\emptyset \subset B \subseteq C$, define-se a linguagem marcada por B como $L_B(A) = \{s \in L(A) : B \cap \chi(\delta(q_0, s)) \neq \emptyset\}$.

5.2.2 Propriedades de AMCs

Um estado $q \in Q$ é acessível se puder ser alcançado por uma seqüência de transições a partir do estado inicial q_0, ou seja, se $\exists s \in \Sigma^*$ tal que $\delta(q_0, s) = q$. Um AMC $A = (Q, \Sigma, C, \delta, \chi, q_0)$ é acessível se todos os seus estados forem acessíveis.

Diz-se que um estado $q \in Q$ é fracamente coacessível e.r.a B se houver uma seqüência de transições que, partindo de q, leve a um estado que marque pelo menos uma cor de B, isto é, se $\exists b \in B$, $\exists s \in \Sigma^*$ tal que $b \in \chi(\delta(q, s))$. A é fracamente coacessível e.r.a B quando algum estado $q \in Q$ for fracamente coacessível e.r.a B.

A é fortemente coacessível e.r.a B se todos os seus estados $q \in Q$ forem fortemente coacessíveis e.r.a B. Um estado $q \in Q$ é fortemente coacessível e.r.a B se, para qualquer cor $b \in B$, houver uma seqüência de transições que, partindo de q, leve a um estado que marque b, isto é, se $\forall b \in B$, $\exists s \in \Sigma^*$ tal que $b \in \chi(\delta(q, s))$.

Diz-se que um AMC A é fracamente *trim* e.r.a B se A for acessível e fracamente coacessível e.r.a B. Da mesma maneira, A é fortemente *trim* e.r.a B se A for acessível e fortemente coacessível e.r.a B.

5.2.3 Operações sobre AMCs

Para a definição formal de algumas operações sobre AMCs, define-se o autômato com marcações coloridas vazio para Σ e C como $\emptyset_{\Sigma,C} = (\emptyset, \Sigma, C, \emptyset, \emptyset, \emptyset)$. Define-se também Q_{ac} como o conjunto de todos os estados acessíveis de Q, $Q_{wco,B}$ como o conjunto de todos os estados de Q que são fracamente coacessíveis e.r.a B e $Q_{sco,B}$ como o conjunto de todos os estados de Q que são fortemente coacessíveis e.r.a B.

A operação $Ac(A)$, que elimina todos os estados não acessíveis de A, é definida por:

$$Ac(A) = \begin{cases} (Q_{ac}, \Sigma, C, \delta|(\Sigma \times Q_{ac}), \chi|(\Sigma \times Q_{ac}), q_0), & se\ Q_{ac} \neq \emptyset \\ \emptyset_{\Sigma,C} \end{cases} \tag{5.1}$$

Define-se $WTr(A, B)$ como a operação sobre A que elimina todos os estados que não são acessíveis e nem fracamente coacessíveis e.r.a B. Seja $Q_{wtr,B} = Q_{ac} \cap Q_{wco,B}$. Então,

$$WTr(A, B) = \begin{cases} (Q_{wtr,B}, \Sigma, C, \delta|(\Sigma \times Q_{wtr,B}), \chi|(\Sigma \times Q_{wtr,B}), q_0), & se\ Q_{wtr,B} \neq \emptyset \\ \emptyset_{\Sigma,C} \end{cases} \tag{5.2}$$

Além disso, define-se $Str(A, B)$ como uma operação sobre A que, em múltiplas iterações, elimina todos os estados que não são acessíveis e nem fortemente coacessíveis e.r.a B. Para isso, define-se $Q_{str,B} = Q_{ac} \cap Q_{sco,B}$ e define-se a função $PStr(A, B)$ como:

$$PStr(A, B) = \begin{cases} (Q_{str,B}, \Sigma, C, \delta|(\Sigma \times Q_{str,B}), \chi|(\Sigma \times Q_{str,B}), q_0), & se\ Q_{str,B} \neq \emptyset \\ \emptyset_{\Sigma,C} \end{cases} \tag{5.3}$$

É natural que, se A for fortemente coacessível e.r.a B, $PStr(A, B)$ também o será. Porém, se não for, $PStr(A, B)$ pode ainda não ser fortemente coacessível e.r.a B, uma vez que esta operação pode apagar estados de $Q - Q_{sco,B}$ que são necessários para a coacessibilidade de alguns estados em $Q_{sco,B}$. Nesse sentido, define-se:

$$\begin{aligned} A_0 &= A \\ A_{j+1} &= PStr(A_j, B),\ j = 0, 1, ... \end{aligned} \tag{5.4}$$

E a operação $Str(A, B)$ como:

$$Str(A, B) = \lim A_j (j \to \infty). \tag{5.5}$$

Para a composição síncrona de dois AMCs, considera-se que, quando uma mesma cor estiver associada a estados de autômatos que modelam subsistemas distintos, a tarefa representada por esta cor deve ser marcada pelo sistema composto apenas quando todos os subsistemas que a possuírem estiverem em estados que a marquem. Assim, da mesma forma que a função de transição composta sincroniza os eventos em comum, a função de marcação composta sincroniza as cores compartilhadas.

Nesse sentido, define-se a composição síncrona de dois AMCs $A_1 = (Q_1, \Sigma_1, C_1, \delta_1, \chi_1, q_{01})$ e $A_2 = (Q_2, \Sigma_2, C_2, \delta_2, \chi_2, q_{02})$ como o AMC:

$$A_1 \| A_2 = Ac(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, C_1 \cup C_2, \delta, \chi, (q_{01}, q_{02})), \tag{5.6}$$

onde

$$\delta((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)), & se\ \sigma \in \Sigma_1 \cap \Sigma_2,\ \sigma \in \Gamma_1(q_1) \cup \Gamma_2(q_2) \\ (\delta_1(q_1, \sigma), q_2), & se\ \sigma \in \Sigma_1,\ \sigma \notin \Sigma_2,\ \sigma \in \Gamma_1(q_1) \\ (q_1, \delta_2(q_2, \sigma)), & se\ \sigma \notin \Sigma_1,\ \sigma \in \Sigma_2,\ \sigma \in \Gamma_2(q_2) \\ indefinida, & caso\ contrario \end{cases} \tag{5.7}$$

$$\chi((q_1, q_2)) = [\chi_1(q_1) \cup (C_2 - C_1)] \cap [\chi_2(q_2) \cup (C_1 - C_2)] \tag{5.8}$$

5.2.4 Bloqueio

A noção de bloqueio num autômato está relacionada com a idéia de se executar uma seqüência de eventos a partir da qual não seja possível completar uma tarefa. Quando o autômato compreende múltiplas cores, a idéia de bloqueio permite duas interpretações, dependendo da exigência de se poder completar tarefas de todas as classes (bloqueio forte) ou de pelo menos uma das classes de tarefas em questão (bloqueio fraco).

Dado um subconjunto não vazio de cores B, diz-se que um autômato $A = (Q, \Sigma, C, \delta, \chi, q_0)$ é fracamente não-bloqueante e.r.a B se $\overline{L_b(A)} = L(A)$, isto é, se qualquer seqüência de eventos gerada for o prefixo de pelo menos uma tarefa completa representada por uma das cores de B.

Dado um subconjunto não vazio de cores B, diz-se que A é fortemente não-bloqueante e.r.a B se $\forall b \in B, \overline{L_b(A)} = L(A)$, isto é, se qualquer seqüência de eventos gerada puder se levada a completar tarefas de todas as classes representadas por cores de B.

O não-bloqueio forte ou fraco de um AMC também pode ser analisado pelas propriedades de seus estados. A é fracamente não-bloqueante e.r.a B se a componente

acessível de A for fracamente coacessível e.r.a B. Já A é fortemente não-bloqueante e.r.a B se a componente acessível de A for fortemente coacessível e.r.a B.

O conjunto de comportamentos fortemente não-bloqueantes e.r.a B contidos em Λ_C, definido por $SNB(\Lambda_C, B) = \{M_C \subseteq \Lambda_C : \forall b \in B, \overline{L_b(M_C)} = \overline{L_C(M_C)}\}$, é fechado para união. Conseqüentemente, ele possui um elemento supremo, denominado $SupSNB(\Lambda_C, B)$, que representa o máximo comportamento fortemente não-bloqueante e.r.a B contido em Λ_C. Seja $A = (Q, \Sigma, C, \delta, \chi, q_0)$ um AMC finito que modela um comportamento colorido regular Λ_C. Então, $SupSNB(\Lambda_C, B) = \Lambda_C(Str(A, B))$. Da mesma maneira, $SupWNB(\Lambda_C, B)$, que representa o máximo comportamento fracamente não-bloqueante e.r.a B contido em Λ_C, é dado por $SupWNB(\Lambda_C, B) = \Lambda_C(WTr(A, B))$.

5.3 Controle Supervisório Multitarefa

Seja uma planta modelada por um AMC $P = (Q, \Sigma, C, \delta, \chi, q_0)$, cujo alfabeto é particionado em eventos controláveis Σ_c e não-controláveis Σ_u. O objetivo do controle supervisório é gerar um supervisor que, desabilitando eventos controláveis, evite que a planta viole um conjunto de condições impostas, as restrições. O supervisor deve também garantir que o sistema controlado seja sempre capaz de completar um determinado conjunto de tarefas relevantes, explicitadas por cores da planta e das restrições. As tarefas relevantes de um SEDMT podem ser expressas através de um conjunto D de cores para o qual se requer não-bloqueio forte.

Como as restrições podem trazer novas informações ao modelo, é conveniente permitir que elas possam definir novos tipos de tarefas. Então, para um conjunto de cores relevantes D, o grupo de restrições impostas à planta, chamado especificação, é representado pelas linguagens admissíveis marcadas por $d \in D$:

$$\begin{aligned} &K_d \subset L_d(P),\ se\ d \in C,\ ou \\ &K_d \subset L(P),\ se\ d \notin C \end{aligned} \tag{5.9}$$

Assim, uma especificação pode ser representada por um comportamento colorido admissível $A_D = \{(K_d, d), d \in D\}$.

Dependendo da relação entre as cores da planta e as cores relevantes, o supervisor pode ter ou não a necessidade de definir a marcação de novos tipos de tarefas. Essa característica diferencia os supervisores incolores e pintores, apresentados nas próximas seções.

5.3.1 Supervisor Incolor

Quando o conjunto de cores de uma especificação está contido nas cores da planta ($D \subseteq C$), as cores do sistema em malha fechada são definidas apenas pela marcação de tarefas da planta. Assim, um supervisor S, neste caso chamado de incolor, consiste simplesmente de uma função $S : L(P) \rightarrow 2^{\Sigma}$ que associa um conjunto de eventos a cada seqüência de eventos gerada pela planta, de forma que os eventos habilitados após a ocorrência de $s \in L(P)$ são dados por $S(s) \cap \Gamma(\delta(q_0, s))$.

Diz-se que um supervisor S é admissível se não implicar a desabilitação de eventos não-controláveis, isto é, se $\forall s \in L(P), \Sigma_u \cap \Gamma(\delta(q_0, s)) \subseteq S(s)$.

Define-se por S/P o SEDMT que representa S controlando P. Com isso, a linguagem gerada por S/P é dada por:

1. $\epsilon \in L(S/P)$;
2. $s\sigma \in L(S/P) \Leftrightarrow (s \in L(S/P)) \wedge (s\sigma \in L(P)) \wedge (\sigma \in S(s))$.

Já as linguagens marcadas por S/P, representando as cadeias marcadas pelo AMC da planta que permanecem após a ação de S, são definidas por:

$$
\begin{aligned}
L_C(S/P) &= L(S/P) \cap L_C(P) \\
\forall c \in C, L_c(S/P) &= L(S/P) \cap L_c(P)
\end{aligned}
\tag{5.10}
$$

Um supervisor incolor S pode ser representado de forma mais sintética por um autômato sem marcação $S_I = (X, \Sigma, \lambda, x_0)$, onde:

- X é o conjunto de estados;
- Σ representa o alfabeto de P;
- $\lambda : \Sigma \times X \rightarrow X$ define a função de transição;
- $x_0 \in X$ é estado inicial.

Assim, se S for admissível, o comportamento do sistema em malha fechada S/P pode ser obtido pela composição síncrona $S_I \| P$, definida por:

$$S_I \| P = Ac(Q \times X, \Sigma, C, \delta_{S_I/P}, \chi_{S_I/P}, (q_0, x_0)), \tag{5.11}$$

onde

$$\delta_{S_I/P}((q,x),\sigma) = \begin{cases} (\delta(q,\sigma),\lambda(q,\sigma)), & se\ \sigma \in \Sigma_1 \cap \Sigma_2,\ \sigma \in \Gamma_1(q_1) \cup \Gamma_2(q_2) \\ (\delta_1(q_1,\sigma),q_2), & se\ \sigma \in \Sigma_1,\ \sigma \notin \Sigma_2,\ \sigma \in \Gamma_1(q_1) \\ (q_1,\delta_2(q_2,\sigma)), & se\ \sigma \notin \Sigma_1,\ \sigma \in \Sigma_2,\ \sigma \in \Gamma_2(q_2) \\ indefinida, & caso\ contrario \end{cases} \quad (5.12)$$

$$\chi_{S_I/P}((q,x)) = \chi(q). \quad (5.13)$$

Observe que esta composição síncrona equivale a usual entre dois AMCs quando todos os estados do supervisor incolor forem marcados por todas as cores de P.

5.3.2 Supervisor Pintor

Muitas vezes as especificações incluem tarefas que não estão explícitas no comportamento em malha aberta da planta ($D - C = B \neq \emptyset$). Nesse caso, a função de marcar novas cores no sistema controlado pertence ao supervisor. Um supervisor pintor é então definido como um mapeamento que associa a cada seqüência de eventos da planta um conjunto de eventos habilitados e um conjunto de novas cores (subconjunto de B) representando tarefas completadas. Assim, um supervisor pintor S consiste de uma função $S : L(P) \to 2^{\Sigma} \times 2^{B}$.

Seja $S(s) = (\gamma, \mu)$. Os eventos que podem ocorrer após a ocorrência de uma cadeia de eventos $s \in L(P)$ são dados por $\gamma \cap \Gamma(\delta(q_0, s))$. Diz-se que um supervisor pintor S é admissível se não implicar a desabilitação de eventos não-controláveis, isto é, se $\forall s \in L(P), \Sigma_u \cap \Gamma(\delta(q_0, s)) \subseteq \gamma$.

A linguagem gerada pelo sistema controlado S/P é definida por:

1. $\epsilon \in L(S/P)$;
2. $s\sigma \in L(S/P) \Leftrightarrow (s \in L(S/P)) \wedge (s\sigma \in L(P)) \wedge (\sigma \in \gamma)$.

Além de marcar com as cores de C as tarefas da planta que permanecem com a supervisão, o sistema controlado S/P marca com cores de B as tarefas definidas pelo supervisor pintor S. Assim, as linguagens marcadas por S/P podem ser descritas por:

$$\begin{aligned} \forall c \in C, L_c(S/P) &= L(S/P) \cap L_c(P); \\ L_C(S/P) &= L(S/P) \cap L_C(P); \\ \forall b \in B, L_b(S/P) &= \{s \in L(S/P) : b \in \mu\}; \\ L_B(S/P) &= \{s \in L(S/P) : \mu \neq \emptyset\}. \end{aligned} \quad (5.14)$$

Um supervisor pintor S pode ser representado por um AMC $S_P = (X, \Sigma, B, \lambda, \kappa, x_0)$, onde:

- X é o conjunto de estados;
- Σ representa o alfabeto de P;
- B é o conjunto de cores novas;
- $\lambda : \Sigma \times X \to X$ define a função de transição;
- $\kappa : X \to 2^B$ define a função de marcação;
- $x_0 \in X$ é estado inicial.

Com isso, caso S seja admissível, o comportamento do sistema em malha fechada S/P pode ser computado pela composição síncrona $S_P \| P$, definida por:

$$S_P \| P = Ac(Q \times X, \Sigma, C \cup B, \delta_{S_P/P}, \chi_{S_P/P}, (q_0, x_0)), \tag{5.15}$$

onde

$$\delta_{S_P/P}((q,x),\sigma) = \begin{cases} (\delta(q,\sigma), \lambda(q,\sigma)), & se\ \sigma \in \Sigma_1 \cap \Sigma_2,\ \sigma \in \Gamma_1(q_1) \cup \Gamma_2(q_2) \\ (\delta_1(q_1,\sigma), q_2), & se\ \sigma \in \Sigma_1,\ \sigma \notin \Sigma_2,\ \sigma \in \Gamma_1(q_1) \\ (q_1, \delta_2(q_2,\sigma)), & se\ \sigma \notin \Sigma_1,\ \sigma \in \Sigma_2,\ \sigma \in \Gamma_2(q_2) \\ indefinida, & caso\ contrario \end{cases} \tag{5.16}$$

$$\chi_{S_P/P}((q,x)) = \chi(q) \cup \kappa(x). \tag{5.17}$$

Ao se levar em consideração que $C \cap B = \emptyset$, nota-se que a composição síncrona apresentada acima equivale a usual entre dois AMCs.

5.3.3 Existência de Supervisores

Seja uma especificação A_D colorida por D ($D - C = B$), tal que, $\forall d \in D \cap C$, $\emptyset \subset L_d(A_D) \subseteq L_d(P)$, e $\forall d \in B$, $\emptyset \subset L_d(A_D) \subseteq L(P)$. As condições necessárias e suficientes para a existência de um supervisor fortemente não-bloqueante e.r.a D tal que $\Lambda_D(S/P) = A_D$ e $L(S/P) = \overline{L_D(A_D)}$ são:

1. o D-fechamento de A_D e.r.a P;

2. a controlabilidade de A_D e.r.a P;

3. o não-bloqueio forte de A_D e.r.a D.

Uma especificação A_D é D-fechada e.r.a P se, para qualquer cor $d \in D \cap C$, toda cadeia de eventos, que seja prefixo de $L_d(A_D)$ e seja marcada por d pela planta, for marcada por d também pela especificação, ou seja, se $\forall d \in (D \cap C), L_d(A_D) = \overline{L_d(A_D)} \cap L_d(P)$.

Um comportamento colorido A_D é controlável e.r.a P se a união de todas as suas linguagens $L_d(A_D)$ forem controláveis, ou seja, se $\overline{L_D(A_D)}\Sigma_u \cap L(P) \subseteq \overline{L_D(A_D)}$. Seja $C(P, A_D) = \{M_D \subseteq A_D : (\overline{L_D(M_D)}\Sigma_u \cap L(P) \subseteq \overline{L_D(M_D)})\}$ o conjunto de linguagens controláveis contidas em A_D. O conjunto $C(P, A_D)$ tem um elemento supremo $SupC(P, A_D)$ [de Queiroz, 2004].

Seja $CSNB(P, A_D, D) = C(P, A_D) \cap SNB(A_D, D)$ o conjunto de linguagens coloridas controláveis e fortemente não-bloqueantes contidas em A_D. O conjunto $CSNB(P, A_D, D)$ tem um elemento supremo $SupCSNB(P, A_D, D)$ [de Queiroz, 2004].

5.3.4 Abordagem Monolítica

A síntese de um supervisor colorido ótimo monolítico é feita com base nos seguintes passos.

1. Modelagem do funcionamento do sistema sem coordenação (planta) e da especificação $A_D \subseteq P$ desejada;

2. Computação do supervisor minimamente restritivo S, cuja linguagem é dada por $SupCSNB(P, A_D, D)$;

3. Eliminação das marcações de S nas cores da planta C.

5.3.5 Abordagem Modular

Seja a planta modelada por um AMC $P = (Q, \Sigma, C, \delta, \chi, q_0)$. Sejam $S_1 : L(P) \to 2^\Sigma \times 2^E$ e $S_2 : L(P) \to 2^\Sigma \times 2^E$ supervisores pintores para P. Para uma cadeia $s \in L(P)$, com $S_1(s) = (\gamma_1, \mu_1)$ e $S_2(s) = (\gamma_2, \mu_2)$, a ação de controle do supervisor $S_1 \wedge S_2 : L(P) \to 2^\Sigma \times 2^E$, representando a conjunção de S_1 e S_2, é dada por $S_1 \wedge S_2 = (\gamma_1 \cap \gamma_2, \mu_1 \cap \mu_2)$.

Um evento controlável é habilitado na planta se e somente se for habilitado por ambos os supervisores para a cadeia correspondente à seqüência de eventos ocorridos na planta. Da mesma forma, as novas cores definidas pelos supervisores são marcadas no sistema em malha fechada se e somente se forem marcadas por ambos os supervisores. Pode-se verificar que o comportamento em malha fechada sob controle modular é dado por:

$$L(S_1 \wedge S_2/P) = L(S_1/P) \cap L(S_2/P); \tag{5.18}$$
$$\forall d \in C \cup E, L_d(S_1 \wedge S_2/P) = L_d(S_1/P) \cap L_d(S_2/P).$$

Em alguns problemas, pode acontecer que o conjunto de novas cores de cada supervisor pintor modular seja diferente. Na ação conjunta, esse fato significa que cada supervisor se preocupa apenas com um subconjunto das novas cores e não impõe qualquer restrição sobre as outras novas cores.

Se S_1 e S_2 forem admissíveis, eles podem ser respectivamente representados por AMCs S_{P1} e S_{P2}, tais que $S_1/P = S_{P1}\|P$ e $S_2/P = S_{P2}\|P$. Então, pela definição de conjunção de supervisores, pode-se verificar que:

$$S_{P1} \wedge S_{P2}/P = S_{P1}\|S_{P2}\|P. \tag{5.19}$$

Para que se possa inferir as propriedades da arquitetura de controle modular a partir das propriedades de cada supervisor, é preciso que se investigue se características como não-bloqueio forte, controlabilidade e D-fechamento são preservadas pela interseção de comportamentos coloridos. As condições para isso são o não-conflito forte e fraco.

Sejam os comportamentos coloridos M_D,$N_D \subseteq 2^{2^{\Sigma^*} \times D}$. Diz-se que M_D e N_D são fracamente não-conflitantes e.r.a $B \subseteq D$ sempre que:

$$\overline{L_B(M_D)} \cap \overline{L_B(N_D)} = \overline{L_B(M_D \cap N_D)}. \tag{5.20}$$

Em palavras, M_D e N_D são fracamente não-conflitantes se cada prefixo comum que puder ser estendido para uma cadeia que complete uma tarefa em M_D, complete a mesma tarefa em N_D. Além disso, diz-se que M_D e N_D são fortemente não-conflitantes quando:

$$\forall b \in B, \overline{L_b(M_D)} \cap \overline{L_b(N_D)} = \overline{L_b(M_D \cap N_D)}. \tag{5.21}$$

Não-conflito forte de M_D e N_D significa que, para cada cor $b \in B$, as linguagens marcadas por essa cor em M_D e em N_D são não-conflitantes ou, em outras palavras, se uma cadeia puder ser estendida para completar uma tarefa em M_D e também puder

ser estendida para completar a mesma tarefa em N_D, existe um modo comum de se completar essa tarefa. Naturalmente, não-conflito forte e.r.a B implica não-conflito fraco e.r.a B.

Sejam, respectivamente, A_M e A_N AMCs para M_D e $N_D \subseteq 2^{2^{\Sigma^*} \times D}$ fracamente não-bloqueantes e.r.a $B \subseteq D$. $A_M \| A_N$ é fracamente não-bloqueante e.r.a B se e somente se M_D e N_D forem fracamente não-conflitantes e.r.a $B \subseteq D$. Além disso, se A_M e A_N forem AMCs fortemente não-bloqueantes. $A_M \| A_N$ é fortemente não-bloqueante e.r.a B se e somente se M_D e N_D forem fortemente não-conflitantes e.r.a B.

Dados dois comportamentos coloridos admissíveis e D-fechados M_D e $N_D \subseteq 2^{2^{\Sigma^*} \times D}$ especificados para um SEDMT P, podem-se calcular supervisores ótimos fortemente não-bloqueantes S_M e S_N, tais que $\Lambda_D(S_M/P) = SupCSNB(P, M_D, D)$ e $\Lambda_D(S_N/P) = SupCSNB(P, N_D, D)$. Se $SupCSNB(P, M_D, D)$ e $SupCSNB(P, N_D, D)$ forem fortemente não-conflitantes, então a conjunção de S_M e S_N é minimamente restritiva e fortemente não-bloqueante, isto é, a arquitetura de controle modular restringe a planta ao mesmo comportamento que um supervisor monolítico ótimo.

5.3.5.1 Abordagem Modular Local

Seja a planta modelada pela composição dos subsistemas do conjunto $\{P_i = (Q_i, \Sigma_i, C_i, \delta_i, \chi_i, q_{0i}), i = 1, 2, ..., n\}$, com $\Sigma_i \cap \Sigma_j = \emptyset$, para $i \neq j$. O alfabeto global é dado por $\Sigma = \cup_{i=1,...,n} \Sigma_i$. Sejam as restrições expressas como um conjunto de comportamentos admissíveis genéricos $\{M_{gen,j} \subseteq 2^{2^{\Sigma_{gen,j}} \times C_{gen,j}}, j = 1, 2, ..., m\}$, com $\Sigma_{gen,j} \subseteq \Sigma$. Para $j = 1, ..., m$, define-se a planta local $P_{loc,j} = (Q_{loc,j}, \Sigma_{loc,j}, C_{loc,j}, \delta_{loc,j}, \chi_{loc,j}, q_{0loc,j})$ associada ao comportamento admissível $M_{gen,j}$, pela composição assíncrona de todos os subsistemas que são afetados por $M_{gen,j}$, isto é, que compartilham eventos com $M_{gen,j}$. Formalmente,

$$P_{loc,j} = \|\{P_i : \Sigma_i \cap \Sigma_{gen,j} \neq \emptyset\}. \tag{5.22}$$

A planta global $P = (Q, \Sigma, C, \delta, \chi, q_0)$ é dada por $P = \|_{j=1,2,...,m} P_{loc,j}$. Para $j = 1, ..., m$, a especificação local $M_{loc,j} \subseteq 2^{2^{\Sigma^*_{loc,j}} \times D_{loc,j}}$ é definida por $M_{loc,j} = M_{gen,j} \| \Lambda_{Cloc,j}(P_{loc,j})$. Define-se $E_{loc,j} = D_{loc,j} - C_{loc,j}$.

Sem perda de generalidade, assume-se que $m = 2$. Sejam $S_{loc,1} : L(P_{loc,1}) \rightarrow 2^{\Sigma_{loc,1}} \times 2^{E_{loc,1}}$ e $S_{loc,2} : L(P_{loc,2}) \rightarrow 2^{\Sigma_{loc,2}} \times 2^{E_{loc,2}}$ supervisores pintores para $P_{loc,1}$ e $P_{loc,2}$, respectivamente. Para $i = 1, 2$, definem-se as projeções naturais $P_{loc,i} : \Sigma^* \rightarrow \Sigma^*_{loc,i}$. Para uma cadeia $s \in L(P)$, com $S_{loc,1}(P_{loc,1}(s)) = (\gamma_1, \mu_1)$ e $S_{loc,2}(P_{loc,2}(s)) = (\gamma_2, \mu_2)$, a ação de controle do supervisor representando a conjunção de $S_{loc,1}$ e $S_{loc,2}$, $S_{loc,1} \wedge S_{loc,2}$:

$L(P) \to 2^{\Sigma} \times 2^{E}$, onde $E = E_{loc,1} \cup E_{loc,2}$, é dada por:

$$S_{loc,1} \wedge S_{loc,2} = ((\gamma_1 \cup (\Sigma_{loc,2} - \Sigma_{loc,1})) \cap (\gamma_2 \cup (\Sigma_{loc,1} - \Sigma_{loc,2})), \quad (\mu_1 \cup (E_{loc,2} - E_{loc,1})) \cap (\mu_2 \cup (E_{loc,1} - E_{loc,2}))). \tag{5.23}$$

A ação de controle dos supervisores locais é estendida para a planta global habilitando-se todos os eventos de Σ que não estão no seu alfabeto local e marcando-se todas as novas cores que não estão no seu conjunto de cores local. Um evento controlável compartilhado é habilitado na planta se e somente se for habilitado pela ação global de ambos supervisores para as cadeias correspondendo às seqüências de eventos ocorridas nas respectivas plantas locais. Da mesma forma, as novas cores comuns aos conjuntos de cores de ambos supervisores são marcadas no comportamento em malha fechada se e somente se for marcada por ambos os supervisores. Pode-se verificar que o comportamento em malha fechada sob controle modular local dado por:

$$L(S_{loc,1} \wedge S_{loc,2}/P) = L(S_{loc,1}/P_{loc,1}) \| L(S_{loc,2}/P_{loc,2});$$

$$\forall d \in D, L_d(S_{loc,1} \wedge S_{loc,2}/P) = \begin{cases} L_d(S_{loc,1}/P_{loc,1}) \| L_d(S_{loc,2}/P_{loc,2}), se \ d \in D_{loc,1} \cap D_{loc,2} \\ L_d(S_{loc,1}/P_{loc,1}) \| L(S_{loc,2}/P_{loc,2}), se \ d \in D_{loc,1} - D_{loc,2} \\ L(S_{loc,1}/P_{loc,1}) \| L_d(S_{loc,2}/P_{loc,2}), se \ d \in D_{loc,2} - D_{loc,1} \end{cases}$$

Portanto,

$$S_{loc,1} \wedge S_{loc,2}/P = (S_{loc,1}/P_{loc,1}) \| (S_{loc,2}/P_{loc,2}). \tag{5.24}$$

Se $S_{loc,1}$ e $S_{loc,2}$ forem admissíveis, eles podem ser respectivamente representados pelos AMCs $A_{loc,1}$ e $A_{loc,2}$ tais que $S_{loc,1}/P_{loc,1} = A_{loc,1} \| P_{loc,1}$ e $S_{loc,2}/P_{loc,2} = A_{loc,2} \| P_{loc,2}$. Então, pela definição anterior de composição de supervisores locais, tem-se:

$$S_{loc,1} \wedge S_{loc,2}/P = A_{loc,1} \| A_{loc,2} \| P. \tag{5.25}$$

Assim, para que as propriedades do sistema controlado na arquitetura modular local sejam deduzidas das propriedades dos supervisores pintores locais, é necessário estudar as condições sob as quais as propriedades de controlabilidade, D-fechamento e não-bloqueio forte são preservadas pela composição síncrona. As condições fundamentais para isso são não-conflito fraco e forte, que são estendidas para comportamentos coloridos com alfabetos locais e conjuntos de cores distintos.

Sejam os comportamentos coloridos $M_{D1} \subseteq 2^{2^{\Sigma_1^*} \times D_1}$ e $N_{D2} \subseteq 2^{2^{\Sigma_2^*} \times D_2}$. Sejam $B \subseteq D = D_1 \cup D_2$, $B_1 = B \cap D_1$ e $B_2 = B \cap D_2$. Diz-se que M_{D1} e N_{D2} são fracamente

não-conflitantes e.r.a B sempre que:

$$\overline{L_{B1}(M_{D1})}\|\overline{L_{B2}(N_{D2})} = L_b(\overline{M_{D1}\|N_{D2}}). \quad (5.26)$$

Em palavras, M_{D1} e N_{D2} são fracamente não-conflitantes se toda cadeia que for um prefixo de M_{D1} e de N_{D2}, quando projetada nos respectivos alfabetos, puder ser estendida para uma cadeia síncrona que complete uma tarefa nos comportamentos em que a tarefa é definida, isto é, se qualquer prefixo síncrono puder completar uma tarefa global. Além disso, diz-se que M_{D1} e N_{D2} são fortemente não-conflitantes e.r.a B quando:

$$\forall b \in B, L_b(\overline{M_{D1}}\|\overline{N_{D2}}) = L_b(\overline{M_{D1}\|N_{D2}}). \quad (5.27)$$

Não-conflito forte de M_{D1} e N_{D2} e.r.a B significa que qualquer prefixo síncrono pode completar qualquer tarefa global representada por uma cor de B. Naturalmente, não-conflito forte e.r.a B implica não-conflito fraco e.r.a B.

Sejam, respectivamente, A_M e A_N AMCs para M_{D1} e N_{D2} fracamente não-bloqueantes e.r.a B_1 e B_2. $A_M\|A_N$ é fracamente não-bloqueante e.r.a B se e somente se M_{D1} e N_{D2} forem fracamente não-conflitantes e.r.a B. Se A_M e A_N forem fortemente não-bloqueantes e.r.a B_1 e B_2. $A_M\|A_N$ é fortemente não-bloqueante e.r.a B se e somente se M_{D1} e N_{D2} forem fortemente não-conflitantes e.r.a B.

Os resultados acima podem ser generalizados para múltiplas especificações utilizando-se as seguintes extensões das definições de não-conflito fraco e forte. Diz-se que um conjunto de comportamentos coloridos $\{M_i \subseteq 2^{2^{\Sigma_i^*} \times D_i}, i = 1, ..., m\}$ é fracamente não-conflitante e.r.a B sempre que

$$\|_{i=1,...,m}\overline{L_{B\cap D_i}(M_i)} = \overline{L_B(\|_{i=1,...,m}M_i)}. \quad (5.28)$$

O conjunto é fortemente não-conflitante e.r.a B quando

$$\forall b \in B, L_b\|_{i=1,...,m}\overline{M_i}) = L_b(\overline{\|_{i=1,...,m}M_i}). \quad (5.29)$$

É possível, portanto, usar uma abordagem modular local para a síntese de supervisores modulares desde que a condição de não-conflito seja satisfeita. Dado um conjunto de AMCs assíncronos que representam o comportamento em malha aberta de uma planta e uma especificação representada por um conjunto de m comportamentos admissíveis genéricos, cada qual sincronizando alguns subsistemas da planta. Temos que:

1. Para $i = 1, ..., m$, obtém-se a planta local $P_{loc,i}$ pela composição de todos os subsistemas que compartilham eventos com o respectivo comportamento admissível

genérico.

2. O conjunto de especificações locais $\{M_{loc,j} \subseteq 2^{2^{\Sigma^*_{loc,j} \times D_{loc,j}}}, i = 1, ..., m\}$ é calculado pela composição de cada comportamento admissível genérico com sua planta local correspondente.

3. Assumindo-se que as especificações locais sejam D-fechadas por construção, pode-se computar supervisores ótimos fortemente não-bloqueantes $S_{loc,i}$, tais que:

$$\Lambda_D(S_{loc,i}/P) = SupCSNB(P_{loc,i}, M_{loc,i}, D_{loc,i}). \tag{5.30}$$

Se o conjunto $\{SupCSNB(P_{loc,i}, M_{loc,i}, D_{loc,i}), i = 1, ..., m\}$ for fortemente não-conflitante e.r.a D, então a conjunção de $S_{loc,i}$ para $i = 1, ..., m$ é minimamente restritiva e fortemente não-bloqueante e.r.a D, isto é, a arquitetura de controle modular local restringe a planta ao mesmo comportamento que um supervisor monolítico ótimo.

5.4 Grail - Módulo Multitarefa

Esta seção apresenta o módulo do Grail para Controle Supervisório que lida com a abordagem multitarefa, o módulo Multitarefa. Todo o desenvolvimento desta foi feito no decorrer deste trabalho.

5.4.1 Classes

Assim como um autômato A pode ser representado por uma máquina de estados finitos, um AMC pode ser representado por uma Máquina de Estados Finitos Colorida (CFM).

Uma CFM é definida por uma quádrupla $FM = (Q_i, \Theta, Q_f, \Theta_f)$, onde Q_i é o conjunto de estados iniciais, Θ é o conjunto de instruções, Q_f é o conjunto de estados finais e Θ_f é o conjunto de instruções finais. Cada instrução final $\theta_f \in \Theta_f$ é composta por:

- Um estado fonte θ_{source}, que indica o estado marcado;
- Uma etiqueta de transição θ_{event}, que representa a cor com a qual o estado θ_{source} é marcado;
- O estado destino θ_{sink}, representando o pseudo-estado FINAL. Este caracteriza a instrução em questão como final.

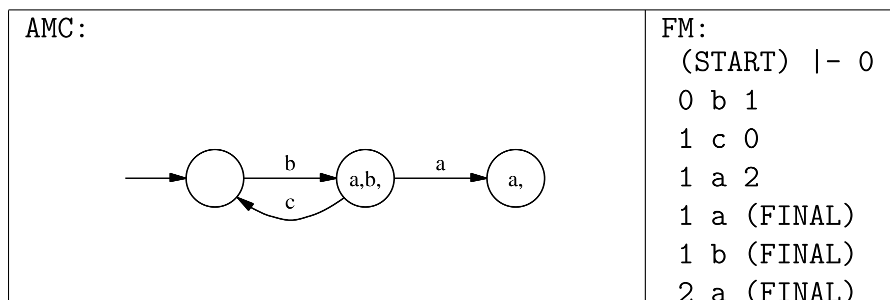


Figura 5.1: Representação de um AMC por uma CFM

Um exemplo é mostrado na Figura 5.1. Neste caso, o conjunto de eventos Σ e o conjunto de cores C são dados, respectivamente, por $\Sigma = \{a, b, c\}$ e $C = \{a, b\}$

São duas as novas classes que geram o AMC. A primeira, `cinst`, define a instrução final e é uma classe derivada de `inst`. A segunda, `cfm`, representa o AMC propriamente dito e é uma classe derivada de `fm`.

5.4.1.1 Colored Finite Machine Final Instructions - cfminst

Gabarito de classe que implementa as instruções finais de uma CFM. Possui três atributos por ser derivado da classe `inst`, que são `source` (θ_{source}), `label` (θ_{event}) e `sink` (θ_{sink}). Entretanto, este necessita apenas de dois atributos, que são:

- θ_{source}: estado ao qual a instrução se refere;
- θ_{event}: que representa a cor que marca o estado em questão.

θ_{sink} é uma constante, cujo valor é o pseudo-estado FINAL. θ_{event} é um parâmetro de tipo a especificar e será especificado de acordo com o tipo de alfabeto utilizado pela CFM.

A criação desta classe implicou, automaticamente, na necessidade de criação das funções de recebimento e impressão de objetos deste tipo. Em outras palavras, a criação das funções *istream* e *ostream* se fizeram necessárias.

5.4.1.2 *Colored Finite Machine* - cfm

Gabarito de classe que implementa as máquinas de estados finitos coloridas. Possui quatro atributos, sendo três deles herdados da classe `fm`. O atributo extra de um CFM representa Θ_f e trata-se de um conjunto não-ordenado de instruções finais que não possui elementos duplicados, ou seja, um Set<CInst<Label>>.

A criação desta classe implicou, automaticamente, na necessidade de criação das funções de recebimento e impressão de objetos deste tipo. Em outras palavras, a criação das funções *istream* e *ostream* se fizeram necessárias.

5.4.2 Filtros

No decorrer deste trabalho foram desenvolvidos 21 filtros para o módulo Multitarefa. Uma breve descrição deles é apresentada a seguir.

5.4.2.1 cfmcolors

Este filtro captura todas as cores de uma CFM passada como parâmetro. Este retorna uma FM que possui apenas um estado e alguns *selfloops*, cujos eventos são as cores do primeiro. Um exemplo é ilustrado na Figura 5.2[1].

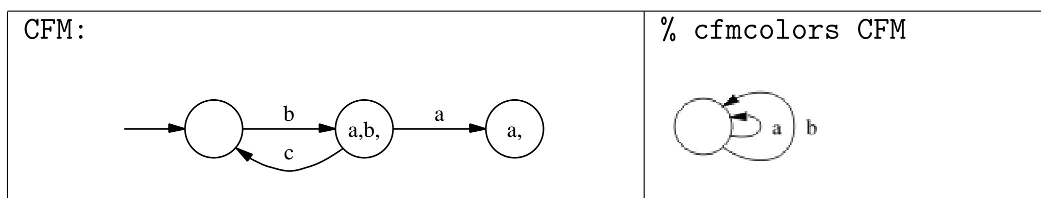


Figura 5.2: Exemplo - cfmcolors

O algoritmo deste filtro consiste nos seguintes passos:

1. Captura dos eventos de todas as instruções finais da CFM, que são as cores da mesma;
2. Criação de uma FM com um estado e um conjunto de instruções que partem deste e chegam neste mesmo estado, cujos eventos são as cores capturadas no passo anterior.

5.4.2.2 cfmcross

Este filtro calcula o produto assíncrono de duas CFMs, sendo que este cálculo gera a intersecção da linguagem das CFMs em questão. Um exemplo é ilustrado na Figura 5.3.

O algoritmo deste filtro consiste nos seguintes passos:

[1]Importante observar que a maioria das CFMs deste capítulo estão representadas graficamente para melhor visualização.

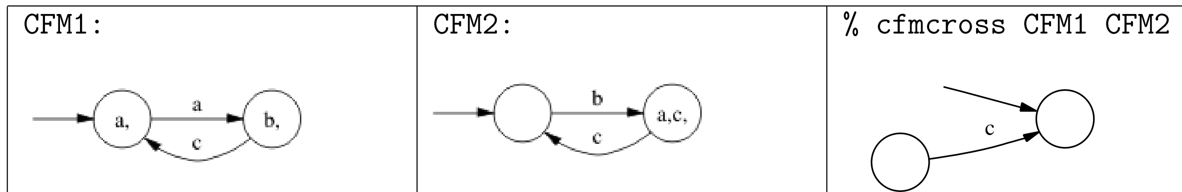


Figura 5.3: Exemplo - cfmcross

1. Captura das CFMs de entrada, `cfm1` e `cfm2`.
2. Cálculo do conjunto de cores C presente em ambas as CFMs.
3. Para cada $c \in C$:
   (a) Computação das FMs `fm1` e `fm2`, sendo que os estados finais são os marcados por c nas CFMs.
   (b) Cálculo do produto assíncrono das FMs acima computadas.
   (c) Computação da CFM, cujos estados iniciais, instruções e estados finais são idênticos ao da FM calculada no passo anterior e suas instruções finais são marcadas por c.
4. Cálculo do produto síncrono entre todas as CFMs computadas no passo anterior.

5.4.2.3 cfmmark

Este filtro marca todos os estados de uma CFM as cores passadas como eventos da FM passada como segundo parâmetro. Um exemplo é ilustrado na Figura 5.4.

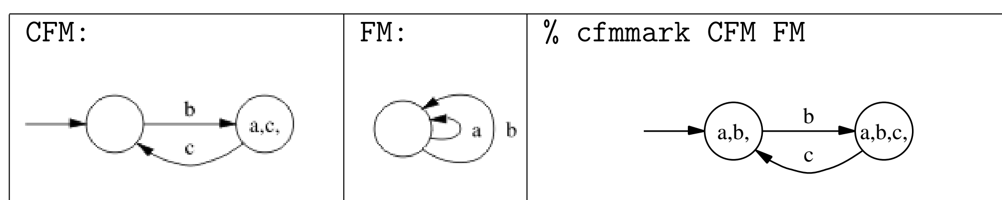


Figura 5.4: Exemplo - cfmmark

O algoritmo deste filtro consiste nos seguintes passos:

1. Captura da CFM e dos eventos da FM passadas como parâmetros.
2. Criação de um conjunto de instruções finais, cujos estados fontes são os estados da CFM e cujos eventos são os da FM.
3. Adição à CFM o conjunto de instruções finais criados no passo anterior.

5.4.2.4 cfmreach

Este filtro computa a componente acessível de uma CFM. Um exemplo é ilustrado na Figura 5.5.

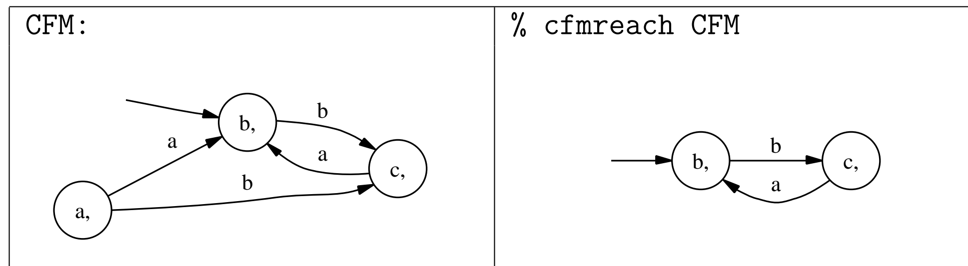


Figura 5.5: Exemplo - cfmreach

O algoritmo deste filtro consiste nos seguintes passos:

1. Captura da CFM.

2. Computação da componente acessível deste.

3. Remoção de todas as instruções finais referentes a estados que não existem mais.

5.4.2.5 cfmrenum

Este filtro renumera uma CFM determinista canonicamente. Um exemplo é ilustrado na Figura 5.6.

CFM:	% cfmrenum CFM
(START) |- 1	(START) |- 0
0 a 1	0 b 1
0 b 2	1 a 0
1 b 2	2 a 0
2 a 1	2 b 1
0 a (FINAL)	0 b (FINAL)
1 b (FINAL)	1 c (FINAL)
2 c (FINAL)	2 a (FINAL)

Figura 5.6: Exemplo - cfmrenum

O algoritmo deste filtro consiste nos seguintes passos:

1. Captura da CFM passada como parâmetro.

2. Criação dos seguintes vetores:

   - **Old:** aonde são armazenados os estados a serem renumerados.

- **New:** aonde é armazenada a nova numeração.

3. Criação da relação entre os estados antigos e sua nova numeração. O estado inicial é renumerado para 0. Os próximos estados a serem renumerados são os alcançáveis a partir do estado inicial por apenas um evento. Este procedimento continua até que todos os estados acessíveis sejam renumerados. Por último, os estados não-acessíveis são renumerados.

4. Substituição dos estados por seu equivalente nas instruções normais e finais.

5.4.2.6 cfmsync

Este filtro computa o produto síncrono entre duas CFMs. Um exemplo é ilustrado na Figura 5.7.

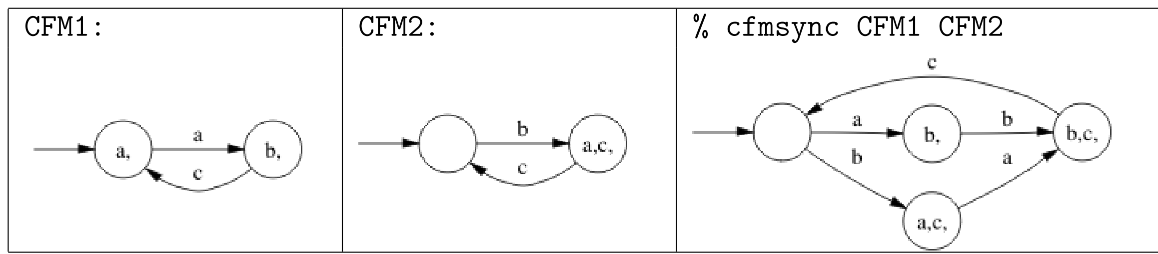


Figura 5.7: Exemplo - cfmsync

O algoritmo deste filtro consiste nos seguintes passos:

1. Captura das duas CFMs de entrada.

2. Computação do produto síncrono entre as FMs provenientes das CFMs, gerando a FM resultante.

3. Transformação da FM resultante na CFM que representa o produto síncrono das CFMs de entrada. Em outras palavras, os estados finais e suas respectivas cores devem ser computados.

   A determinação das cores de cada estado resultante é proveniente da Equação 5.8. Para identificar que estados compõem cada estado da FM resultante, utiliza-se da Equação $q = q_1 + n_1.q_2$ apresentada no Capítulo 4.

5.4.2.7 cfmstrim

Este filtro computa a componente acessível e fortemente coacessível e.r.a B de uma CFM passada como parâmetro. Os eventos da FM de entrada caracterizam as cores pertencentes a B. Um exemplo de utilização deste filtro é ilustrado na Figura 5.8.

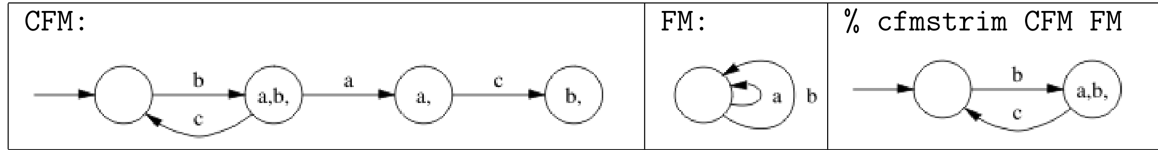


Figura 5.8: Exemplo - cfmstrim

O algoritmo deste filtro consiste nos seguintes passos:

1. Captura da CFM e dos eventos da FM, o conjunto de cores B.
2. Verificação das propriedades de acessibilidade e forte coacessibilidade e.r.a B da CFM. Em caso positivo, o algoritmo é encerrado.
3. Para cada $b \in B$, define-se uma FM cujos estados marcados são os que são marcados por b. Computa-se a componente acessível e coacessível de uma FM, considerando o resultado como entrada para o cálculo da próxima FM. O resultado final deste procedimento caracteriza a CFM acessível e coacessível e.r.a B.
4. Remoção das instruções finais referentes a estados não mais existentes.
5. Retorno ao segundo passo.

Observe que este algoritmo é iterativo porque a eliminação de determinados estados pode eliminar a forte coacessibilidade e.r.a B de outros estados.

5.4.2.8 cfmwtrim

Este filtro computa a componente acessível e fracamente coacessível e.r.a B de uma CFM passada como parâmetro. Os eventos da FM de entrada caracterizam as cores pertencentes a B. Um exemplo de utilização deste filtro é ilustrado na Figura 5.9.

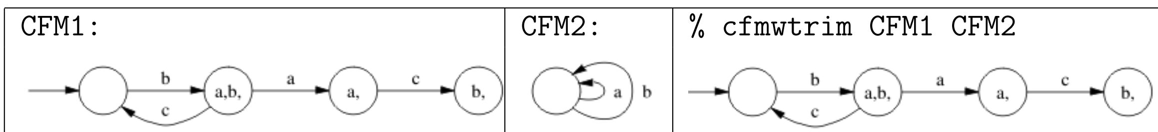


Figura 5.9: Exemplo - cfmwtrim

O algoritmo deste filtro consiste nos seguintes passos:

1. Captura da CFM e dos eventos da FM, o conjunto de cores B.
2. Computação da componente acessível e coacessível e.r.a B da FM, cuja linguagem é dada por $L_m(FM) = L_B(CFM)$.

3. Remoção das instruções finais referentes a estados não mais presentes na CFM em questão.

5.4.2.9 cfmcondat

Dadas duas CFMs, uma planta e um comportamento colorido desejado (A), este filtro computa todos os eventos da planta que são efetivamente desabilitados por A. Um exemplo é mostrado na Figura 5.10.

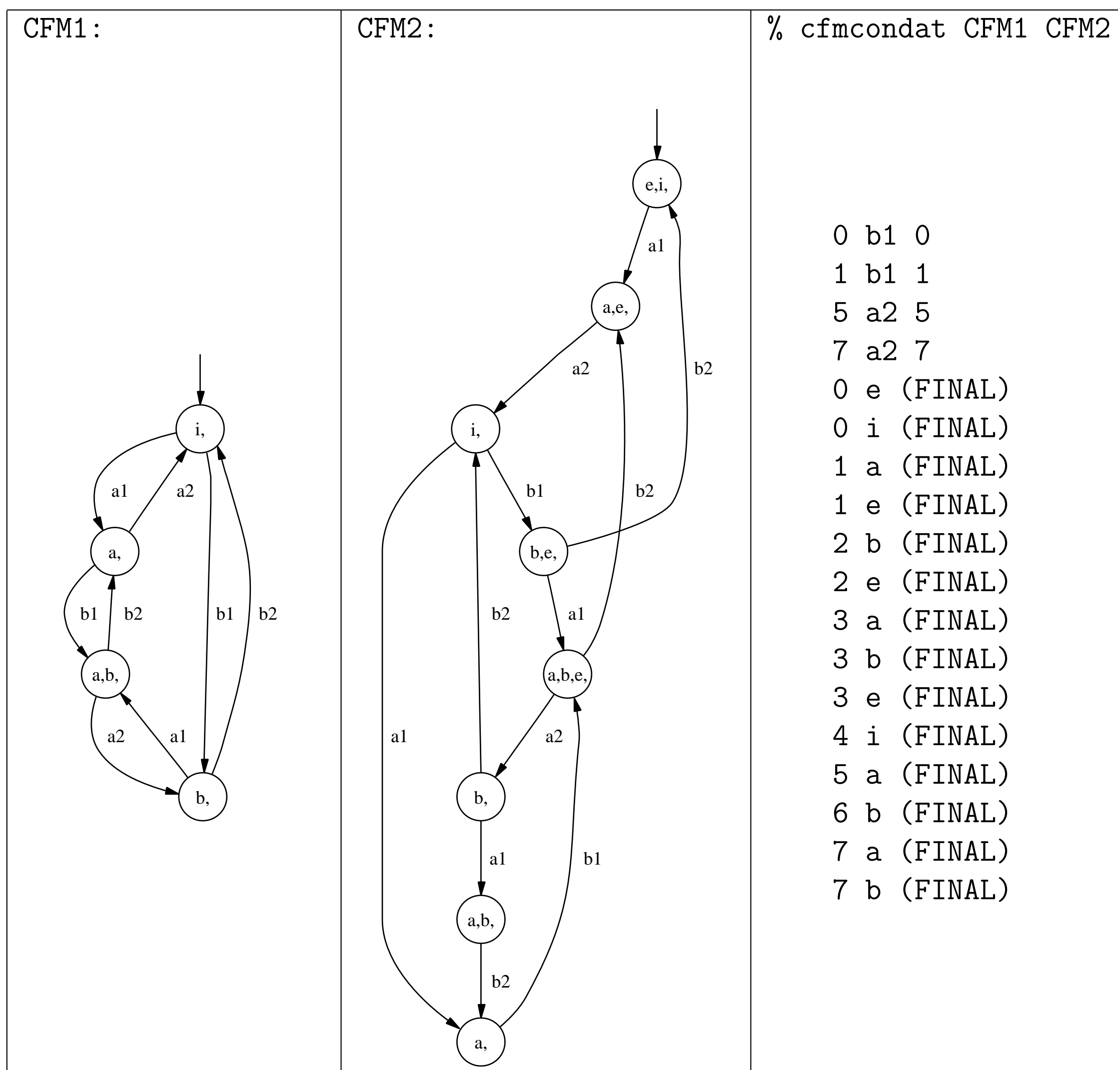


Figura 5.10: Exemplo - cfmcondat

O algoritmo deste filtro consiste nos seguintes passos:

1. Computação do supervisor através do cálculo do produto assíncrono entre a planta e o comportamento colorido desejado e da captura da componente acessível da resultante.

2. Para cada estado do supervisor, deve-se:

   (a) Encontrar os estados da planta e do comportamento desejado referentes ao estado em questão.

   (b) Capturar os eventos desabilitados, ou seja, os eventos ausentes no estado do supervisor e presentes no estado equivalente da planta.

   (c) Criar instruções caracterizadas por *selfloops*, cujos estados são definidos pelos equivalentes do comportamento colorido e cujos eventos são os desabilitados.

3. Retorna as instruções finais criadas no passo anterior como saída.

5.4.2.10 cfmsupcsnb

O objetivo deste filtro é a computação do supremo comportamento colorido controlável e fortemente não-bloqueante de uma planta P em relação a um determinado comportamento colorido desejado A.

A sintaxe deste é dada por: `% cfmsupcsnb P A ncont D`, sendo P a planta, A o comportamento colorido desejado, *ncont* o conjunto de eventos não-controláveis e D o conjunto de cores relevantes.

O algoritmo implementado segue a orientação apresentada em [de Queiroz, 2004]. Seja $P = (Q, \Sigma, C, \delta, \chi, q_0)$ a planta e $H_a = (X, \Sigma, D, \lambda_a, \kappa_a, x_0)$ um AMC fracamente *trim* que modela o comportamento colorido desejado A. Seja P' um autômato sem marcação que gere $L(P)$, define-se $H = H_a \| P'$ com $H = (X \times Q, \Sigma, D, \lambda, \kappa, (x_0, q_0))$. O algoritmo proposto para calcular o supremo comportamento colorido controlável e fortemente não-bloqueante contido em A consiste de uma eliminação iterativa de estados de H_0 que não sejam controláveis e fortemente *trim* e.r.a D até se convergir para o maior ponto fixo.

Conseqüentemente, os passos que compõem este algoritmo são:

1. Captura dos parâmetros passados.

2. Computação de H_a, P' e H. Faz-se $H' = H$.

3. Computação do supremo comportamento colorido controlável contido em H'.

4. Computação do supremo comportamento colorido fortemente *trim* e.r.a D contido no resultado do passo anterior.

5. Verificação da convergência do algoritmo.

- Se o algoritmo convergiu, o algoritmo pode ser encerrado e o resultado obtido no passo 4 é o supremo comportamento colorido controlável e fortemente não-bloqueante e.r.a D contido em A.
- Se o algoritmo não convergiu, faz-se H' igual ao resultado obtido no passo 4 e retorna-se o passo 3.

5.4.2.11 cfmsupcwnb

O objetivo deste filtro é a computação do supremo comportamento colorido controlável e fracamente não-bloqueante de uma planta P em relação a um determinado comportamento colorido desejado A.

A sintaxe deste é dada por: `% cfmsupcwnb P A ncont D`, sendo P a planta, A o comportamento colorido desejado, *ncont* o conjunto de eventos não-controláveis e D o conjunto de cores relevantes.

O algoritmo em questão tem como base o utilizado no filtro `cfmsupcsnb`, apresentado na seção anterior. Conseqüentemente, os passos que compõem este algoritmo são:

1. Captura dos parâmetros passados.
2. Computação de H_a, P' e H. Faz-se $H' = H$.
3. Computação do supremo comportamento colorido controlável contido em H'.
4. Computação do supremo comportamento colorido fracamente *trim* contido no resultado do passo anterior.
5. Verificação da convergência do algoritmo.
   - Se o algoritmo convergiu, o algoritmo pode ser encerrado e o resultado obtido no passo 4 é o supremo comportamento colorido controlável e fracamente não-bloqueante e.r.a D contido em A.
   - Se o algoritmo não convergiu, faz-se H' igual ao resultado obtido no passo 4 e retorna-se o passo 3.

5.4.2.12 cfmsupred

Este filtro é um redutor de supervisores. Trata-se de uma adaptação do `fmsupred` para AMCs. Sua sintaxe é dada por: `% cfmsupred cfm1 cfm2`, sendo que `cfm1` representa a planta e `cfm2` o supervisor.

Para a computação das coberturas de controle, é importante observar que dois estados não podem pertencer ao mesmo bloco se, entre outras propriedades, um deles for marcador e outro desmarcador. Este é o único passo do algoritmo de redução de supervisores que a marcação tanto da planta quanto do supervisor são levados em consideração.

O algoritmo do `fmsupred` cria, antes da computação das coberturas de controle, um vetor de estados proibidos aonde ficam listados todos os pares compostos por um estado marcador e um desmarcador. No caso do `cfmsupred`, fixa-se cada uma das cores da planta $c \in C$ e verifica-se a consistência de marcação entre os estados da planta e do supervisor para a cor em questão.

5.4.2.13 cfmtofm

Este filtro transforma uma CFM em uma FM. Seu algoritmo consiste na captura da CFM de entrada e na computação de uma FM, cujos atributos Q_i, Θ e Q_f são idênticos ao da CFM de entrada. Um exemplo é ilustrado na Figura 5.11.

```
CFM:
    (START) |- 1
    0 a 1
    0 b 2
    1 b 2
    2 a 1
    0 a (FINAL)
    1 b (FINAL)
    1 c (FINAL)
```

```
% cfmtofm CFM
    (START) |- 1
    0 a 1
    0 b 2
    1 b 2
    2 a 1
    0 |- (FINAL)
    1 |- (FINAL)
```

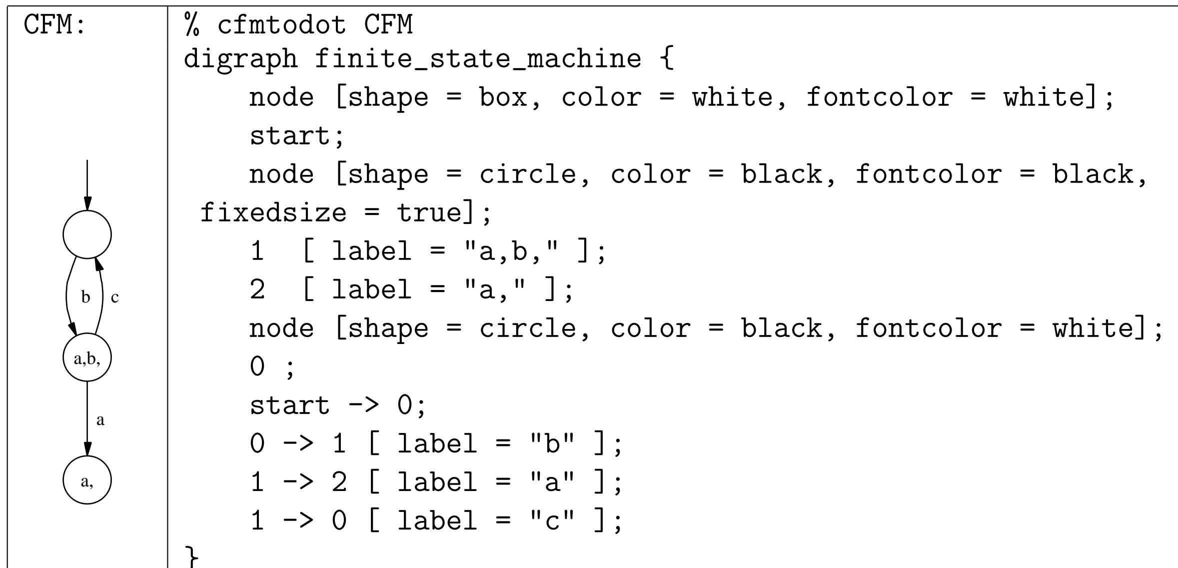
Figura 5.11: Exemplo - cfmtofm

5.4.2.14 cfmtodot

Este filtro converte uma CFM no formato aceito pelo Graphviz. O algoritmo deste filtro consiste na impressão da características da CFM de entrada no texto pré-determinado pelo formato Graphviz. Um exemplo é ilustrado na Figura 5.12.

5.4.2.15 cfmiscontrol

Este filtro tem como objetivo a verificação da controlabilidade de um comportamento colorido desejado em relação à planta. Sua sintaxe é dada por: `%cfmiscontrol`




```
% cfmtodot CFM
digraph finite_state_machine {
    node [shape = box, color = white, fontcolor = white];
    start;
    node [shape = circle, color = black, fontcolor = black,
 fixedsize = true];
    1  [ label = "a,b," ];
    2  [ label = "a," ];
    node [shape = circle, color = black, fontcolor = white];
    0 ;
    start -> 0;
    0 -> 1 [ label = "b" ];
    1 -> 2 [ label = "a" ];
    1 -> 0 [ label = "c" ];
}
```

Figura 5.12: Exemplo - cfmtodot

`cfm1 cfm2 fm3`, sendo `cfm1` a planta, `cfm2` o comportamento colorido desejado e `fm3` o conjunto de eventos não-controláveis.

O algoritmo deste filtro foi baseado no `cfmcondat` e consiste nos seguintes passos:

1. Computação do supervisor através do cálculo do produto assíncrono entre a planta e o comportamento colorido desejado e da captura da componente acessível da resultante.

2. Para cada estado do supervisor, deve-se:

   (a) Encontrar os estados da planta e do comportamento desejado referentes ao estado em questão.

   (b) Capturar os eventos desabilitados e comparar os mesmos com os eventos não-controláveis. Se algum dos eventos desabilitados for não-controlável, o algoritmo é encerrado e `The specification is NOT controllable with respect to the plant`.

3. Se o algoritmo não foi encerrado antes deste passo, então `The specification is controllable with respect to the plant`.

5.4.2.16 cfmisdclosed

A sintaxe deste filtro é dada por `% cfmisdclosed cfm1 cfm2 fm3`, sendo que `cfm1` representa a planta P, `cfm2` o comportamento colorido desejado A e `fm3` o conjunto de cores D. Este filtro verifica se A é D-fechado em relação à P.

O algoritmo deste filtro consiste nos seguintes passos:

1. Capturar os parâmetros de entrada.
2. Para cada cor $d \in D$:
   (a) Computar os autômatos que representam as linguagens marcadas $L_d(P)$ e $L_d(A)$, P' e A', respectivamente.
   (b) Calcular o produto síncrono $S = P' \| A'$.
   (c) Marcar todos os estados de S e computar produto síncrono deste com P'.
   (d) Se os resultados obtidos em `b` e `c` não forem isomorfos, encerrar o algoritmo e `The desired colored behaviour is NOT D-closed wrt the plant`.
3. Se o algoritmo não foi encerrado antes deste passo, então `The desired colored behaviour is D-closed wrt the plant`.

5.4.2.17 cfmisomorph

Este filtro verifica se duas CFMs deterministas são isomorfas. O algoritmo deste filtro consiste nos seguintes passos:

1. Capturar as CFMs passadas como parâmetro.
2. Renumerar canonicamente ambas as CFMs.
3. Comparar todas as instruções das CFMs.
   - Se todas as instruções forem idênticas, então `The CFMs are isomorphic.`
   - Se existir pelo menos uma instrução diferente, então `The CFMs are NOT isomorphic.`

5.4.2.18 cfmisstrim

Este filtro verifica se uma CFM é fortemente coacessível e.r.a um conjunto de cores B. Um exemplo é ilustrado na Figura 5.13.

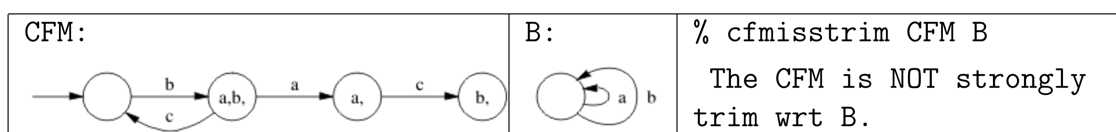


Figura 5.13: Exemplo - cfmisstrim

O algoritmo deste filtro consiste em:

1. Capturar a CFM e do conjunto de cores B.
2. Para cada cor $b \in B$:
   (a) Computar a FM que represente $L_b(CFM)$.
   (b) Verificar a acessibilidade e coacessibilidade deste autômato. Se ele não for *trim*, então o algoritmo é encerrado e `The CFM is NOT strongly trim wrt B`.
3. Se o algoritmo não foi encerrado antes deste passo, então `The CFM is strongly trim wrt B`.

5.4.2.19 cfmiswtrim

Este filtro verifica se uma CFM é fracamente coacessível e.r.a um conjunto de cores B. Um exemplo é ilustrado na Figura 5.14.

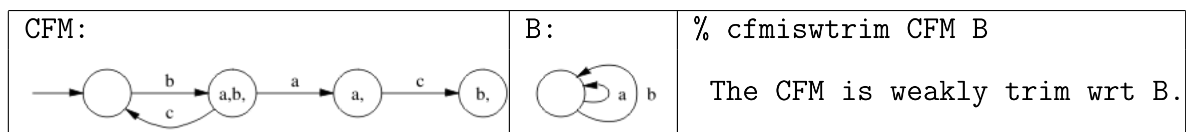


Figura 5.14: Exemplo - cfmiswtrim

O algoritmo deste filtro consiste em:

1. Capturar a CFM e do conjunto de cores B.
2. Computar a FM que represente $L_B(CFM)$.
3. Verificar a acessibilidade e coacessibilidade deste autômato. Se ele for *trim*, então `The CFM is weakly trim wrt B`. Caso contrário, `The CFM is NOT weakly trim wrt B`.

5.4.2.20 cfmissnconf

Este filtro verifica se duas ou mais CFMs são fortemente não-conflitantes e.r.a um conjunto de cores B. O algoritmo deste filtro consiste em:

1. Capturar todas as CFMs passadas como parâmetros, assim como o conjunto de cores B.
2. Computar o produto síncrono entre todas as CFMs.
3. Verificar se a resultante é fortemente *trim* e.r.a B.

- Em caso positivo, `The CFMs are NOT strongly conflicting`.
- Em caso negativo, `The CFMs are strongly conflicting`.

5.4.2.21 cfmiswnconf

Este filtro verifica se duas ou mais CFMs são fracamente não-conflitantes e.r.a um conjunto de cores B. O algoritmo deste filtro consiste em:

1. Capturar todas as CFMs passadas como parâmetros, assim como o conjunto de cores B.
2. Computar o produto síncrono entre todas as CFMs.
3. Verificar se a resultante é fracamente *trim* e.r.a B.
   - Em caso positivo, `The CFMs are NOT weakly conflicting`.
   - Em caso negativo, `The CFMs are weakly conflicting`.

5.5 Exemplo - Labirinto do Gato e do Rato

Para elucidar as idéias apresentadas no decorrer deste capítulo, uma extensão do exemplo do labirinto do gato e do rato é introduzida. Nesta extensão, o gato e o rato compartilham o labirinto presente na Figura 5.15. Na cozinha há alimento disponível para o gato e para o rato. Espera-se obter um controlador que garanta a maior liberdade de movimento evitando que ambos animais ocupem o mesmo recinto ao mesmo tempo de forma que o gato e o rato sempre possam comer (tarefas `g` e `r`, respectivamente) e o sistema possa voltar ao estado inicial (tarefa `i`).

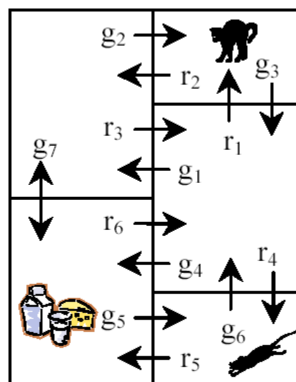


Figura 5.15: Labirinto do Gato e do Rato

Sejam o alfabeto $\Sigma = \{g_i, r_j : 1 \leq i \leq 7, 1 \leq j \leq 6\}$ e o conjunto de cores $C = \{i, g, r\}$. Modela-se o movimento do gato e do rato pelo sistema composto pelos autômatos P_g e P_r sobre Σ e C, apresentados nas Figuras 5.16 e 5.17, de forma que a planta global P seja obtida por $P = P_g \| P_r$.

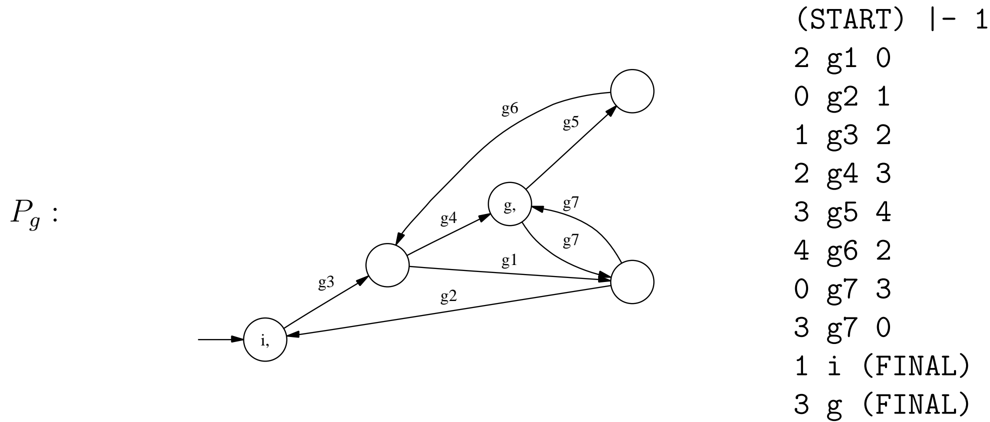


Figura 5.16: Modelo Obtido para o Gato

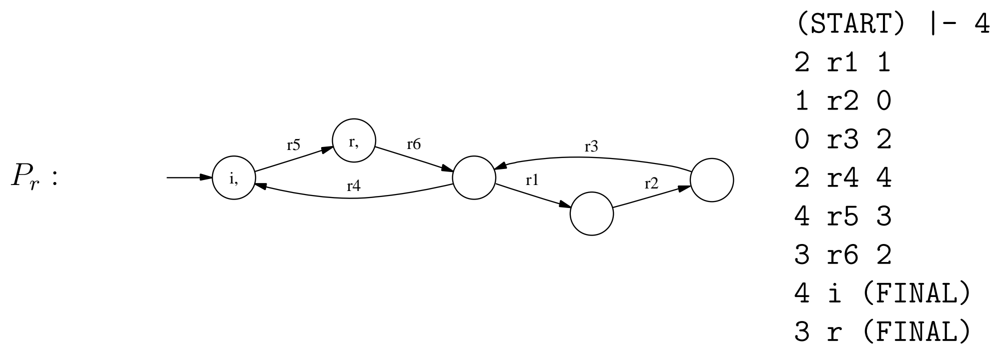


Figura 5.17: Modelo Obtido para o Rato (P_r)

Um autômato que represente o máximo comportamento A_C e que respeite as especificações é obtido de P, apagando-se os estados em que os dois animais ocupem o mesmo recinto. A especificação A_C também pode ser obtida a partir de um conjunto de restrições R_i, cujo objetivo é impedir a entrada de um animal no recinto i ocupado pelo outro.

O máximo comportamento controlável e fortemente não bloqueante e.r.a $\{i, g, r\}$ contido em A_C é, então, computado e apresentado na Figura 5.18. Eliminando-se todas as cores desse modelo, obtém-se um supervisor incolor S que serve de solução ótima para esse problema de controle.

Os passos realizados para a resolução deste exemplo através dos filtros do Grail para Controle Multitarefa foram:

```
% cfmsync Pg Pr > P
% cfmsync R0 R1 | cfmsync R2 | cfmsync R3 | cfmsync R4 > Ac
```

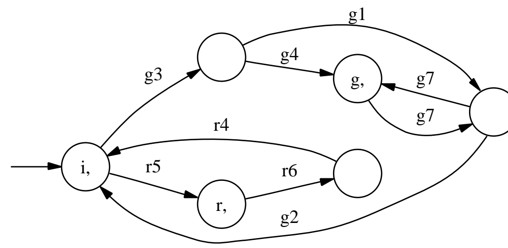


Figura 5.18: Supervisor Obtido para o Labirinto do Gato e do Rato

```
% type ncont
    0 g7 0
% type B
    0 i 0
    0 g 0
    0 r 0
% cfmsupcsnb P Ac ncont B > S
```

5.6 Conclusões

Este capítulo introduziu a Teoria de Controle Supervisório Multitarefa que tem como base os Autômatos com Marcação Colorida. O objetivo deste capítulo é a apresentação do módulo Multitarefa, descrevendo a sua estrutura e os seus filtros. O desenvolvimento deste módulo foi realizado no decorrer deste trabalho e esta se mostra satisfatória para a resolução de exemplos que envolvem Controle Supervisório Multitarefa.

Capítulo 6

Controle Supervisório de SEDs com Marcação Flexível

Um Sistema a Eventos Discretos com Marcação Flexível (SEDMF) é um sistema descrito por uma linguagem e uma estrutura de controle. A linguagem descreve o conjunto de todas as palavras que o SED pode gerar sobre seu alfabeto. A estrutura de controle é uma função que associa cada palavra do sistema a um conjunto de controles, onde define-se um possível conjunto de eventos habilitados e a marcação ou não da palavra em questão. Em outras palavras, num SEDMF a estrutura de controle é determinada de tal forma que, para cada palavra gerada pelo sistema, tem-se, além da ação de habilitação dos eventos, o atributo de marcação da palavra. Este capítulo apresenta, baseado em [da Cunha, 2003], os SEDMFs e a extensão da teoria de controle supervisório associada a estes.

[da Cunha, 2003] disponibilizou filtros que lidavam com esta nova abordagem e que utilizavam o Grail como uma biblioteca de funções. Uma das contribuições deste trabalho é a criação do módulo que lida com a abordagem explorada por [da Cunha, 2003], o módulo Hierárquico. Esta é composta por novas classes que possibilitam o tratamento de SEDMFs como objetos comuns ao Grail e contém os filtros desenvolvidos em [da Cunha, 2003] e adaptados à nova estrutura do Grail para Controle Supervisório.

6.1 Noções Preliminares

Um SEDMF P sobre o alfabeto Σ é um par (L_P, Γ_P), onde $L_P \subseteq \Sigma^*$ é uma linguagem prefixo-fechada e Γ_P é uma função que associa cada palavra $s \in L_P$ a um conjunto de controles $\Gamma_P(s) \subseteq 2^{\Sigma} \times \{M, N\}$. A linguagem L_P representa o conjunto de todas as palavras em Σ que P pode gerar e, para cada palavra $s \in L_P$, $\Gamma_P(s)$ é um

conjunto de controles $(\gamma, \#)$, sendo γ o conjunto de eventos habilitados após s e $\#$ um indicador de marcação. Se $\#$ for igual a M, a palavra s é considerada marcada e, se $\#$ for igual a N, a palavra s é considerada não-marcada.

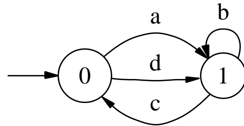
Sobre os controles que compõem Γ_P define-se algumas operações. Dados $(\gamma_1, \#_1)$ e $(\gamma_2, \#_2)$ em $2^\Sigma \times \{M, N\}$, temos:

- **Ordem Parcial $\subseteq$:** $(\gamma_1, \#_1) \subseteq (\gamma_2, \#_2)$ se e somente se $\gamma_1 \subseteq \gamma_2$ e $\#_1 \leq \#_2$. Convém citar que $\leq$ é uma operação para o conjunto $\{M, N\}$, tal que $N \leq M$, $N \leq N$ e $M \leq M$.

- **União** $\cup$: $(\gamma_1, \#_1) \cup (\gamma_2, \#_2) = (\gamma_1 \cup \gamma_2, \#_1 \wedge \#_2)$. Convém citar que $\wedge$ é uma operação para o conjunto $\{M, N\}$, tal que $N \wedge N = N$, $N \wedge M = N$, $M \wedge N = N$ e $M \wedge M = M$.

Dado o SEDMF P sobre o alfabeto Σ, representa-se L_P por um autômato $A = (Q, \Sigma, \delta, q_0, Q_m)$, de forma que $L_A = L_P$ e $L_{A,m} = \emptyset$. Em outras palavras, para todo $s \in L_P$, deve existir $q \in Q$ tal que $\hat{\delta}^1(q_0, s) = q$ e $Q_m = \emptyset$. Representa-se a estrutura de controle Γ por uma tabela cujas entradas relacionam as palavras de P aos seus respectivos conjuntos de controle. Assim, o par (A, Γ) é uma representação em estados do SEDMF P. Um exemplo é apresentado na Figura 6.1.

A:



Γ:

Palavras	Controles
ϵ	$(\emptyset, M)(\{a\}, M)$
$((a+d)b^*c)^*(a+d)$	$(\{c, b\}, N)$
$((a+d)b^*c)^+$	$(\emptyset, N)(\{a\}, N)(\{a, d\}, M)$

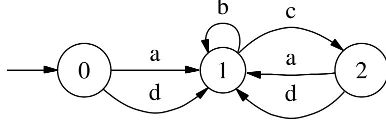
Figura 6.1: Exemplo de um SEDMF P Representado pelo Par (A, Γ)

Alternativamente representa-se o SEDMF P sobre o alfabeto Σ pelo par (A', Γ'), onde $A' = (Q', \Sigma, \delta', q'_0, Q'_m)$ é novamente um autômato e Γ' é uma função que associa cada estado $q \in Q'$ a um conjunto de controles $\Gamma'(q)$. O par (A', Γ') deve ser tal que $L_{A'} = L_P$, $L_{A',m} = \emptyset$ e, para toda palavra $s \in L_P$ e estado $q \in Q'$ tais que $\hat{\delta}'(q_0, s) = q$, deve-se ter $\Gamma'(q) = \Gamma_P(s)$. Em outras palavras, ao se associar P ao par (A', Γ'), P é representado por um autômato e uma estrutura de controle dependente do estado. O SEDMF da Figura 6.1 pode ser representado por intermédio de um autômato cuja estrutura de controle é dependente dos estados, Figura 6.2.

Os modelos para a maioria dos SEDs dotados de controle encontrados na literatura podem ser expressos como casos particulares do modelo para SEDMF. Na abordagem

[1] $\hat{\delta}$ é a extensão da função δ para palavras em Σ^*

A':



Γ':

Estados	Controles
0	$(\emptyset, M)(\{a\}, M)$
1	$(\{c, b\}, N)$
2	$(\emptyset, N)(\{a\}, N)(\{a, d\}, M)$

Figura 6.2: Exemplo de um SEDMF P Representado pelo Par (A', Γ')

de Ramadge e Wonham, um SED P sobre o alfabeto Σ é representado pelo par de linguagens (L, L_m), sendo L a linguagem gerada e L_m a marcada. O mecanismo de controle é definido pela partição de Σ em subconjuntos de eventos controláveis Σ_c e não-controláveis Σ_u. A tradução do sistema P para um SEDMF é feita da seguinte forma:

- $L_P = L$;
- $\forall s \in L_P,\ \Gamma_P(s) = \{(\gamma, \#) \in 2^\Sigma \times \{M, N\} : (\Sigma_u \subseteq \gamma)\ e\ (\# = M,\ se\ s \in L_m)\ e\ (\# = N,\ se\ s \in (L - L_m)\}$.

6.2 Controle Supervisório de SEDMFs

Para o SEDMF P sobre o alfabeto Σ, um supervisor S é uma função $S : L_P \rightarrow 2^\Sigma \times \{M, N\}$ que associa a cada palavra $s \in L_P$ um controle $S(s) \in \Gamma_P(s)$. Para a palavra $s \in L_P$, se $S(s) = (\gamma, \#) \in \Gamma_P(s)$, o conjunto ativo de eventos em P após s, $\Sigma_{L_P}(s)$, fica restrito a $\gamma \cap \Sigma_{L_P}(s)$ e considera-se s como marcada apenas se $\# = M$.

O sistema em malha fechada, denotado S/P, possui comportamento definido por duas linguagens:

- A linguagem gerada, $L_{S/P}$, definida recursivamente por:
  - $\epsilon \in L_{S/P}$;
  - Para $s \in L_P$ e $\sigma \in \Sigma$, $s\sigma \in L_{S/P}$ se e somente se $s\sigma \in L_P$, $s \in L_{S/P}$ e $\sigma \in \gamma$, com $S(s) = (\gamma, \#)$.
- A linguagem marcada, $L_{S/P,m}$, definida por:
  - $s \in L_{S/P,m}$ se e somente se $s \in L_{S/P}$ e $S(s) = (\gamma, M)$.

A linguagem gerada em malha fechada $L_{S/P}$ contém as palavras em L_P permitidas por S sob supervisão e é prefixo-fechada. A linguagem marcada em malha fechada

$L_{S/P,m}$, é uma linguagem contida em $L_{S/P}$ onde, para todas as palavras, o supervisor escolhe o indicador M. De forma geral, $\overline{L_{S/P,m}} \subseteq L_{S/P}$, isto é, pode haver palavras geradas em malha fechada que não sejam prefixos de palavras marcadas em malha fechada. Tais palavras não evoluem para completar uma tarefa do sistema, caracterizando a situação de bloqueio. Portanto, para o SEDMF P e o supervisor S, diz-se que S é não-bloqueante se $\overline{L_{S/P,m}} = L_{S/P}$.

Dados o SEDMF P e a linguagem-alvo $K \subseteq L_P$, o problema de controle supervisório consiste em encontrar um supervisor não-bloqueante S para P tal que $\emptyset \subset L_{S/P,m} \subseteq K$. A existência de um supervisor não-bloqueante S para P tal que $L_{S/P,m} = K$ é garantida se e somente se K for (L_P, Γ_P)-compatível.

Uma linguagem K é (L_P, Γ_P)-compatível se e somente se:

- Para todo $s \in K$, existir (γ, M) em $\Gamma_P(s)$ tal que $\gamma \cap \Sigma_{L_P}(s) = \Sigma_K(s)$;
- Para todo $s \in \overline{K} - K$, existir (γ, N) em $\Gamma_P(s)$ tal que $\gamma \cap \Sigma_{L_P}(s) = \Sigma_K(s)$.

Para um SEDMF P e uma linguagem-alvo $K \subseteq L_P$, existe a possibilidade de K não ser (L_P, Γ_P)-compatível e, conseqüentemente, não existir um supervisor não-bloqueante para P que realize o comportamento especificado por K. Neste caso, existe um conjunto de linguagens (L_P, Γ_P)-compatíveis contidas em K, cujo elemento supremo é único e denotado por $supC_{(L_P,\Gamma_P)}(K)$. Este elemento caracteriza o comportamento compatível e minimamente restritivo ao comportamento de P que segue à especificação K.

O problema de controle supervisório possui solução, portanto, se e somente se $supC_{(L_S,\Gamma_S)}(K) \supset \emptyset$. Emprega-se, portanto, a máxima linguagem (L_P, Γ_P)-compatível contida em K para implementar o supervisor que segue a K de forma ótima.

6.2.1 Exemplo - Um Gato e um Rato num Labirinto

Considere a extensão do problema clássico do gato e do rato num labirinto apresentada na Figura 6.3. O labirinto em questão consiste em quatro salas, onde o gato e o rato estão confinados. O gato está inicialmente na sala 1 e o rato na sala 3. Os animais deslocam-se de uma sala a outra através de uma rede privada de dutos, representados pelas linhas que conectam as salas da Figura 6.3.

Os dutos são direcionais e o acesso do animal ao duto pode ser proibido por uma porta. Em alguns casos, o fechamento de um duto acarreta no fechamento de outro duto. Por exemplo, se o rato estiver na sala 3, o fechamento do acesso à sala 1 proíbe também o acesso à sala 2. Para se manterem no labirinto, os animais devem ser capazes de se alimentar e a comida está disponível no interior de alguns dutos, como indicado

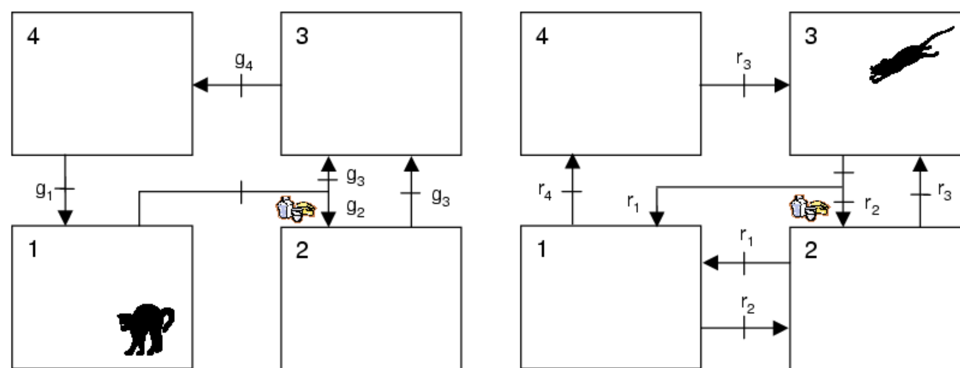
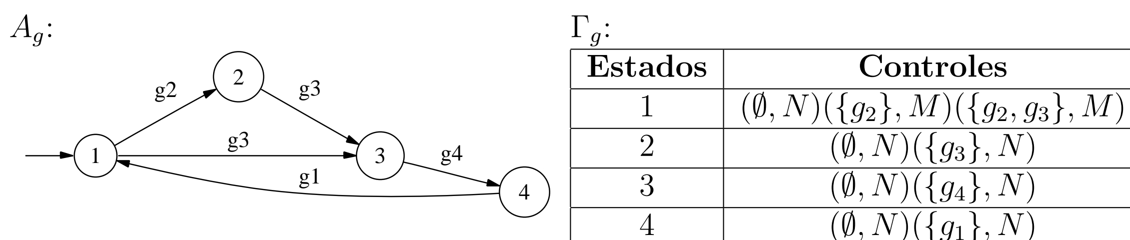


Figura 6.3: Labirinto do Gato e do Rato

na Figura 6.3. Quando o acesso a um duto é proibido, o acesso à comida presente no mesmo é interrompido.

O problema de controle supervisório a ser tratado é projetar um supervisor para manter o gato e o rato vivos no labirinto. O supervisor não deve permitir a presença de ambos os animais na mesma sala e o acesso individual à comida deve ser garantido para que os animais não morram de fome.

Os comportamentos do gato e do rato são modelados pelos SEDMFs apresentados nas Figuras 6.4 e 6.5. Os eventos g_i e r_j indicam que o gato e o rato entraram nas salas i e j, respectivamente, para $i \in \{1,2,3,4\}$ e $j \in \{1,2,3,4\}$. A numeração dos estados dos autômatos indica a sala ocupada.



Estados	Controles
1	$(\emptyset, N)(\{g_2\}, M)(\{g_2, g_3\}, M)$
2	$(\emptyset, N)(\{g_3\}, N)$
3	$(\emptyset, N)(\{g_4\}, N)$
4	$(\emptyset, N)(\{g_1\}, N)$

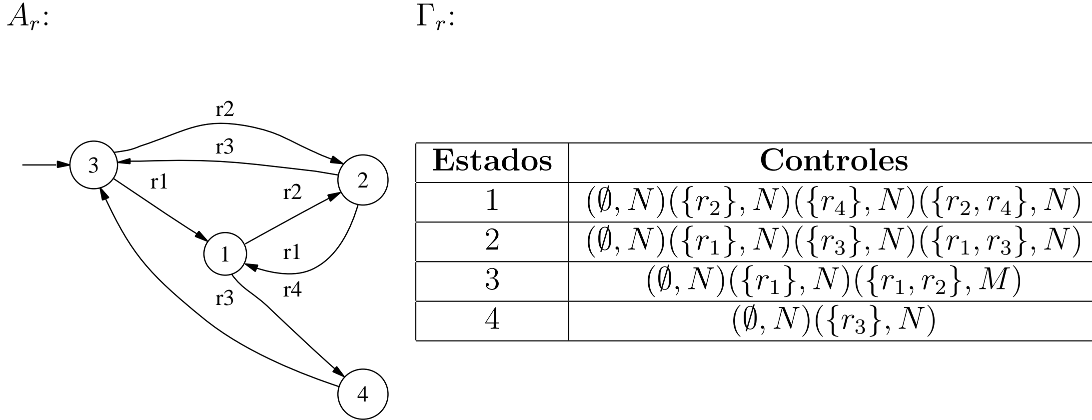
Figura 6.4: Modelo Obtido para o Gato

Os conjuntos de controles para o gato e o rato são construídos por observação das opções de controle e marcação no labirinto da Figura 6.3. Quando o duto que sai de uma sala pode ser fechado, o correspondente evento pode ser retirado de um controle disponível para o estado que corresponde à referida sala. Quando o acesso a um duto com comida é permitido por um controle, considera-se este controle como marcado.

O comportamento conjunto do gato e do rato no labirinto é representado por um SEDMF $P = (A, \Gamma)$, onde A é um autômato e Γ é uma estrutura de controle dependente do estado. O SEDMF P é obtido pela composição síncrona dos SEDMFs que representam o comportamento do gato e do rato individualmente de forma que:

- $A = A_g \| A_r$, sendo que A possui 16 estados e 48 transições.

A_r: Γ_r:



Estados	Controles
1	$(\emptyset, N)(\{r_2\}, N)(\{r_4\}, N)(\{r_2, r_4\}, N)$
2	$(\emptyset, N)(\{r_1\}, N)(\{r_3\}, N)(\{r_1, r_3\}, N)$
3	$(\emptyset, N)(\{r_1\}, N)(\{r_1, r_2\}, M)$
4	$(\emptyset, N)(\{r_3\}, N)$

Figura 6.5: Modelo Obtido para o Rato

- O conjunto de controles $\Gamma(i, j)$ é composto por elementos $(\gamma, \#)$ tal que γ contém um possível conjunto de eventos permitidos de ocorrer após o estado (i, j). Se γ contém um evento que corresponde à permissão do acesso à comida, então $\# = M$. Caso contrário, $\# = N$.

  Por exemplo, para o estado $(1, 3)$ de P o controle correspondente é:

$$\Gamma(1,3) = \{(\emptyset, N), (\{r_1\}, N), (\{r_1, r_2\}, M), (\{g_2\}, M), (\{g_2, r_1\}, M),$$
$$(\{g_2, r_1, r_2\}, M), (\{g_2, g_3\}, M), (\{g_2, g_3, r_1\}, M), (\{g_2, g_3, r_1, r_2\}, M)\}.$$

A linguagem-alvo $K \subseteq L_A$ é obtida a partir da eliminação dos estados de A que correspondem à ocupação do gato e do rato na mesma sala, da marcação dos estados que dão acesso à comida e da obtenção do componente *trim* do autômato resultante. Determina-se a máxima linguagem (A, Γ)-compatível contida em K, apresentada na Figura 6.6. O estado 3 representa ocupação das salas 1 e 3 pelo gato e pelo rato, respectivamenente. Assim como o estado 15 representa a ocupação das salas 4 e 3. Estes estados correspondem aos controles $(\{g_2\}, M)$ e $(\{r_1, r_2, g_1\}, M)$ e permitem o acesso à comida para o gato e o rato, sendo conseqüentemente marcados.

6.3 Grail - Módulo Hierárquico

Esta seção apresenta o módulo do Grail para Controle Supervisório que lida com SEDMFs, denominado Hierarquico. Todo o desenvolvimento desta, a exceção dos filtros que compõem a mesma, são contribuições deste trabalho.

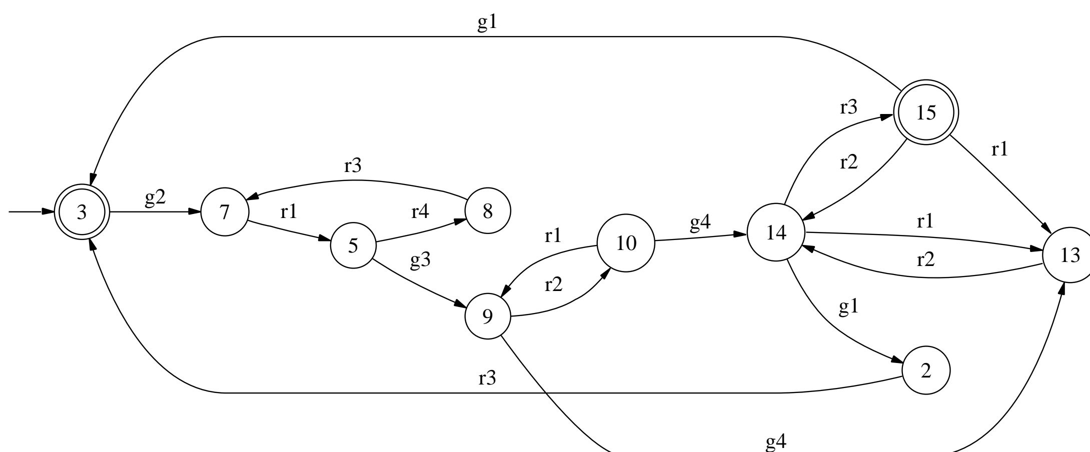
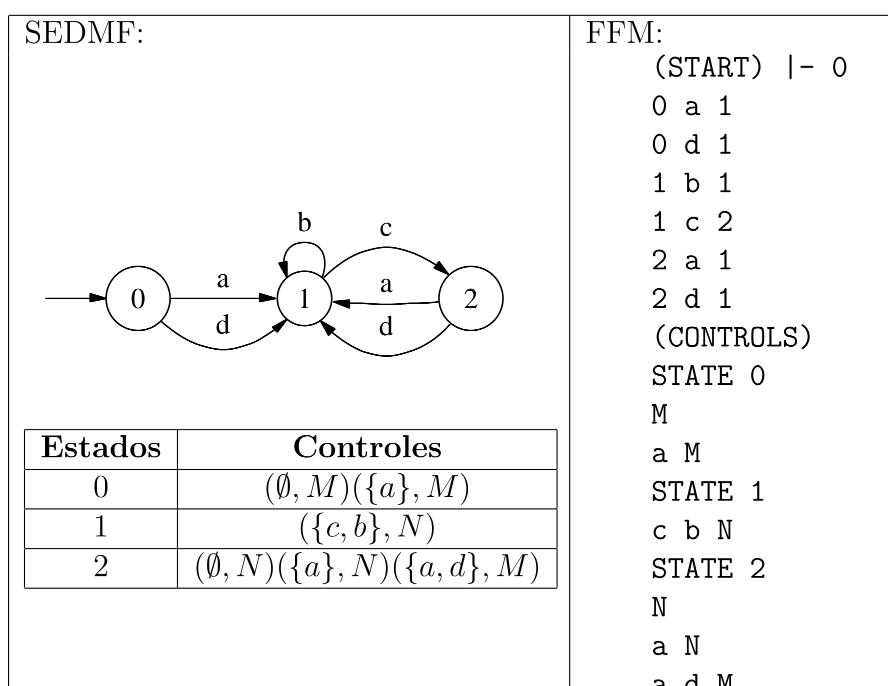


Figura 6.6: Máxima Linguagem (A, Γ)-Compatível

6.3.1 Classes

Assim como um autômato pode ser representado por uma máquina de estados finitos, um SEDMF pode ser representado por uma Máquina de Estados Finitos Flexível (FFM).

Uma FFM é definida por uma quádrupla $FFM = (Q_i, \Theta, Q_f, Q_\Gamma, \Gamma)$, onde Q_i é o conjunto de estados iniciais, Θ é o conjunto de instruções, Q_f é o conjunto de estados finais, Q_Γ é o conjunto de estados associados aos conjuntos de controles e Γ representa os conjuntos de controles associados a cada Q_Γ. Um exemplo é mostrado na Figura 6.7.



Estados	Controles
0	$(\emptyset, M)(\{a\}, M)$
1	$(\{c, b\}, N)$
2	$(\emptyset, N)(\{a\}, N)(\{a, d\}, M)$

FFM:

```
(START) |- 0
0 a 1
0 d 1
1 b 1
1 c 2
2 a 1
2 d 1
(CONTROLS)
STATE 0
M
a M
STATE 1
c b N
STATE 2
N
a N
a d M
```

Figura 6.7: Representação de um SEDMF por uma FFM

São três as novas classes que geram o SEDMF, sendo elas apresentadas nas seções seguintes.

6.3.1.1 Control - ctrl

Classe que implementa os controles de uma FFM. Possui dois atributos, que são:

- γ: conjunto de eventos habilitados;
- $\#$: indicador de marcação.

A criação desta classe implicou, automaticamente, a necessidade de criação de alguns métodos, tais como:

- **Geração:** cria um determinado controle com base em alguns atributos passados como parâmetro.
- **Intersecção:** dados os controles $ctrl1$ e $ctrl2$, temos que $ctrl1 \cap ctrl2 = (\gamma_1 \cap \gamma_2, \#_1 \vee \#_2)$.
- **Comparação:** os operadores $<$,$>$,$\leq$ e $\geq$ seguem as regras da ordem parcial. Dados os controles $ctrl1$ e $ctrl2$, temos que $ctrl1 < ctrl2$ se e somente se $\gamma_1 \subset \gamma_2$ e $\#_1 < \#_2$.
- **Igualdade:** o operador $==$ compara dois controles verificando a sua igualdade, enquanto que $=$ associa ao primeiro controle os atributos do segundo.
- **União:** dados os controles $ctrl1$ e $ctrl2$, temos que $ctrl1 \cup ctrl2 = (\gamma_1 \cup \gamma_2, \#_1 \wedge \#_2)$.

Além disso, foi necessária a implementação das funções de recebimento e impressão dos objetos pertencentes a esta classe. São elas: *istream* e *ostream*, respectivamente.

6.3.1.2 Control Set - ctrl_set

Classe que implementa os conjuntos de controles de uma FFM. Esta classe derivada de `set` não possui nenhum atributo extra. Foi criada devido à peculiaridade de algumas operações sobre conjuntos de controles.

Os métodos implementados para esta classe são:

- **Geração:** computa o conjunto de controles possíveis de existir num autômato, tendo como base o conjunto de eventos não-controláveis.
- **Intersecção:** realiza a intersecção entre os controles pertencentes a dois conjuntos diferentes.
- **Adição:** o operador $+=$ adiciona a um conjunto de controles o segundo objeto, podendo este ser um controle ou conjunto de controles.
- **Pertence:** verifica se um conjunto de controles contém um determinado controle ou um determinado conjunto de controles.
- **União:** realiza a união entre os controles pertencentes a dois conjuntos diferentes.

As funções *istream* e *ostream* de recebimento e impressão, respectivamente, dos objetos pertencentes a esta classe também foram implementadas.

6.3.1.3 Flexible Finite Machine - ffm

Classe que implementa as FFMs propriamente ditas. Possui cinco atributos, sendo três provenientes de `fm`. Os atributos extras pertencentes à esta classe são:

- **Estrutura de Controle (Γ):** trata-se de uma instanciação da classe `array` para a classe `ctrl_set`. Cada elemento deste atributo caracteriza o conjunto de controles que deve ser associado a um determinado estado da FFM.
- **Index (Q_Γ):** associa cada elemento da estrutura de controle a um determinado estado da FFM.

6.3.2 Filtros

Os filtros presentes neste módulo são:

1. `ffmcondat` - encontra as etiquetas de transição de uma FFM desabilitados por uma segunda FM.
    - **Sintaxe:** `ffmcondat ffm fm`
    - **Descrição:** São dois os parâmetros deste filtro, sendo `ffm` a planta e `fm` o autômato que modela a linguagem-alvo.
2. `ffmcross` - computa o produto cartesiano entre duas FFMs.

   - **Sintaxe:** `ffmcross ffm1 ffm2`

3. `ffmdeterm` - calcula uma FFM determinista a partir da FFM passada como parâmetro.

   - **Sintaxe:** `ffmdeterm ffm1`

4. `ffmloop` - faz o *selfloop* de eventos numa FFM.

   - **Sintaxe:** `fmloop ffm fm`
   - **Descrição:** Cria, para cada estado da `ffm`, transições do estado para ele mesmo, *selfloops*, envolvendo todos os eventos pertencentes ao alfabeto da `fm`.

5. `ffmmin` - minimiza o número de estados de uma FFM para duas classes de equivalência: Nerode (mesma extensão para palavras) e Gamma (mesmo conjunto de controle).

   - **Sintaxe:** `ffmmin ffm1`

6. `ffmreach` - computa a componente acessível de uma FFM.

   - **Sintaxe:** `ffmreach ffm1`

7. `ffmsupc` - computa a máxima linguagem compatível para uma planta que segue uma determinada especificação.

   - **Sintaxe:** `ffmsupc ffm1 fm2`
   - **Descrição:** São dois os parâmetros deste filtro, sendo `ffm1` a planta e `fm2` a FM que modela a linguagem-alvo.

8. `ffmsync` - calcula a composição síncrona de duas FFMs.

   - **Sintaxe:** `ffmsync ffm1 ffm2`

As funções *istream* e *ostream* de recebimento e impressão, respectivamente, dos objetos pertencentes a esta classe também foram implementadas.

6.4 Conclusões

Este capítulo apresentou, de forma sucinta, a Teoria de Controle Supervisório para SEDs com Marcação Flexível. O objetivo deste capítulo é a introdução do módulo Hieraquico, descrevendo a sua estrutura e familiarizando o leitor com os seus filtros. A contribuição do presente trabalho à este módulo é a estruturação da mesma, facilitando o seu uso e a implementação de novos filtros.

Capítulo 7

Sistemas Condição/Evento

Sistemas Condição/Evento (SCE) são SEDs que possuem duas classes de sinais:

- Os sinais condição, constantes por partes e que tomam valores de um conjunto finito de condições;
- Os sinais eventos, não-nulos apenas em pontos discretos do tempo e que assumem valores sobre um conjunto finito de eventos.

Em geral, ambos os tipos de sinais podem ser sinais de entrada e saída para SCEs. Formalmente, transições de estados podem ser habilitadas ou desabilitadas pelo sinal de entrada condição e forçadas pelo sinal de entrada evento.

A abordagem de SCEs se adapta perfeitamente à modelagem de:

- Sistemas a Eventos Discretos, tornando a modelagem destes mais natural e intuitiva à oferecida por formalismos estritamente discretos;
- Sistemas Híbridos, possibilitando que a modelagem destes seja baseada em diagramas de blocos e fluxos de sinais.

Este capítulo apresenta a abordagem de SCEs para a modelagem de SEDs e de sistemas híbridos, assim como a extensão da teoria de controle supervisório para esta abordagem, sendo estas seções baseadas em [Rodrigues, 2004]. Além disso, o módulo Sistemas Condição/Evento desenvolvido no Grail para Controle Supervisório é apresentada.

7.1 Modelagem de SCEs

Dados os conjuntos finitos e disjuntos U,V,Y e Z. Um SCE em $[0,\infty)$ com entrada condição u, entrada evento v, saída condição y e saída evento z, é um mapa $\mathcal{S}$: $U \times V \rightarrow 2^{Y \times Z}$ tal que, para cada entrada $(u(\cdot), v(\cdot)) \in U \times V$ existe pelo menos uma saída $(y(\cdot), z(\cdot)) \in Y \times Z$ tal que $(y(\cdot), z(\cdot)) \in \mathcal{S} : (u(\cdot), v(\cdot))$.

Uma realização de estados discreta para um SCE $\mathcal{S} : U \times V \rightarrow 2^{Y \times Z}$ é a quíntupla (Q, δ, g, h, q_0), onde:

- Q é um conjunto enumerável de estados;
- $\delta : Q \times U \times V \rightarrow 2^Q - \emptyset$ é a função de transição de estado, satisfazendo $q \in \delta(q, u, 0), \forall q \in Q$ e $\forall u \in U$. Ou seja, o sistema deve estar apto a permanecer em qualquer estado se não houverem sinais de entrada evento;
- $g : Q \times U \rightarrow Y$ é a função de saída condição;
- $h : Q \times Q \times V \rightarrow Z$ é a função de saída evento, satisfazendo $h(q, q, 0) = 0$ para $q \in Q$. Ou seja, a saída evento só será diferente de 0 se houver uma transição de estado;
- q_0 é o estado inicial.

Uma vez que um SCE possui sinais de entrada e saída que assumem valores discretos no tempo, é possível associar a este um comportamento lógico, o qual será representado por linguagens de palavras de comprimento finito. A linguagem de um SCE é definida pelas seqüências dos valores que os sinais eventos e condições assumem em seus pontos de descontinuidade, sem registrar no entanto os instantes de ocorrência de tais descontinuidades.

Como sinais condição são atualizados somente nos instantes de ocorrência de eventos, associa-se um evento fictício, denominado evento de inicialização e denotado por η para possibilitar a representação da aplicação de um sinal condição antes da ocorrência de qualquer sinal evento. Como η não está associado a transições de estado da planta, logo $\eta \notin V$. O conjunto $V^+ = \{\eta\} \cup V$ denota a inclusão do evento η em V.

O comportamento lógico de um SCE pode ser descrito por um par de linguagens:

- A linguagem gerada, que representa o comportamento fisicamente possível de ocorrer no sistema em termos de entrada e saída;
- A linguagem marcada, que corresponde às cadeias do sistema que possuem algum significado especial.

Os SCEs $\mathcal{S} : U \times V \rightarrow 2^{Y \times Z}$ podem, a partir de suas linguagens, ser modelados por autômatos. O autômato que representa um SCE é definido por uma quíntupla $(Q, \Sigma, \delta, Q_m, q_0)$, onde:

- Q é o conjunto de estados finitos;
- $\Sigma \subseteq (V^+ \times U)$ caracteriza o alfabeto;
- $\delta : Q \times \Sigma \rightarrow 2^Q$ é a função de transição de estado, possivelmente parcial e não-determinista;
- Q_m representa o conjunto de estados marcados;
- q_0 é o estado inicial.

A Figura 7.1 ilustra um exemplo simples de um autômato que modela o comportamento seqüêncial de um SCE, onde $\Sigma = \{\eta u_1, v_2 u_2, v_3 u_3\}$.

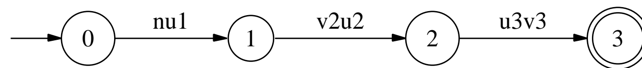


Figura 7.1: Exemplo - Representação de um SCE por um Autômato

7.2 SCEs como Modelos para SEDs

Os SCEs definem claramente sinais de entrada e saída, mostrando-se bastantes interessantes para modelar SEDs coordenados por CLPs. Baseado nisto, apresenta-se uma metodologia de modelagem para SEDs através da resolução de uma simplificação do exemplo da célula de manufatura, Seção 2.4.2.

O primeiro passo na modelagem de um sistema consiste em encontrar os subsistemas envolvidos no sistema como um todo. São quatro os subsistemas envolvidos neste exemplo: a mesa, o alimentador, o enchedor e o manipulador robótico. No entanto, apenas o modelo da mesa é aqui apresentado.

A mesa possui uma realização de estado discreta $S_{mesa} = (Q_{mesa}, f_{mesa}, h_{mesa}, q_{mesa}(0))$ com os conjuntos V_{mesa} e U_{mesa}, onde:

- Q_{mesa} é o conjunto de estados $Q_{mesa} = \{m_g, m_p\}$;
- V_{mesa} é o conjunto de saída evento $V_{mesa} = \{v_{mesa}\}$;
- U_{mesa} é o conjunto de entradas condição $U_{mesa} = \{m^{on}, m^{off}\}$;

- f_{mesa} é a função de transição $f_{mesa} : Q_{mesa} \times U_{mesa} \to Q_{mesa}$ e está definido na Tabela 7.1;
- h_{mesa} é a função de transição $h_{mesa} : Q_{mesa} \times U_{mesa} \to V^{+}_{mesa}$ e está definido na Tabela 7.2;
- $q_{mesa(0)} = m_p$ é o estado inicial da mesa.

q_{mesa}	u_{mesa}	q_{mesa}
m_g	m^{on}	m_g
m_g	m^{off}	m_p
m_p	m^{on}	m_g
m_p	m^{off}	m_p

Tabela 7.1: Função de Transição da Mesa

x^{-}_{mesa}	x_{mesa}	v_{mesa}
m_g	m_g	0
m_g	m_p	v_{mesa}
m_p	m_g	0
m_p	m_p	0

Tabela 7.2: Função de Saída Evento da Mesa

A mesa pode encontrar-se em dois estados distintos, girando (m_g) ou parada (m_p). A entrada condição m^{on} simboliza o comando de ligar a mesa e m^{off} o comando de desligar a mesa. O evento v_{mesa} sinaliza o fim do giro da mesa. A Figura 7.2 mostra o autômato que representa o comportamento lógico seqüencial da mesa.

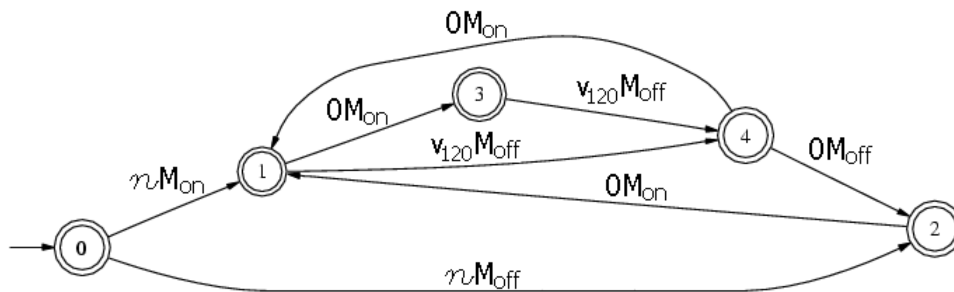


Figura 7.2: Autômato que Representa a Mesa

Após a obtenção do modelo condição/evento de cada um dos subsistemas da célula de manufatura, pode-se obter o modelo para a planta livre P. Este modelo é obtido através do algoritmo da operação empilhar, o qual foi apresentado em [Garcia, 2002]. Este algoritmo retorna as funções de transição e saída evento da planta livre, bem como uma tabela de tradução, sendo possível identificar os conjuntos de estados, de entradas condição e de saídas evento.

Após obter as funções de transição e saída evento é possível encontrar um autômato que represente a linguagem do sistema. Isto é possível através do algoritmo para encontrar o autômato condição/evento, que também foi implementado em [Garcia, 2002]. Este algoritmo encontrou um autômato com 17 estados e 496 transições para representar o comportamento da planta livre.

7.3 Controle Supervisório de SCE

Esta seção apresenta a teoria de controle supervisório para SCE e baseia-se nos resultados apresentados em [Rodrigues, 2004].

7.3.1 Abordagem Monolítica

Sem perda de generalidade, podemos considerar uma estrutura de controle que se utiliza de um modelo simplificado, conforme mostra a Figura 7.3. Nesta estrutura, a planta possui entrada condição e saída evento, enquanto que o supervisor possui entrada evento e saída condição.

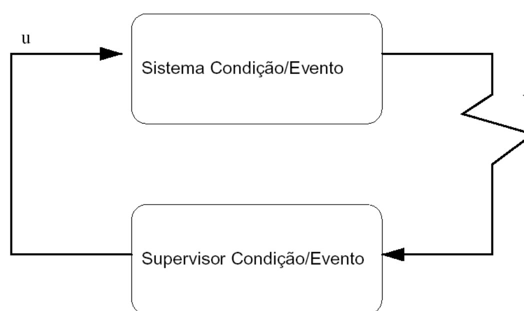


Figura 7.3: Estrutura de Controle

Os sinais evento na planta forçam transições simultâneas no supervisor e sinais condição do supervisor habilitam ou desabilitam transições de estado na planta. Sendo assim, sinais eventos v fluem da planta para o supervisor, enquanto que os sinais condição u fluem do supervisor para a planta. O sinal evento v assume valores sobre um conjunto discreto de eventos V e o sinal condição u assume valores sobre um conjunto discreto de condições U.

A planta possui uma realização de estados discreta $P = (Q, f, h, q_0)$ com os conjuntos U e V, conforme definido anteriormente. O supervisor pode ser definido como uma realização de estados discreta $S = (Y, \xi, \psi, y_0)$ com os conjuntos V e U - os mesmos da planta a ser controlada, onde:

- Y é o conjunto de estados;
- $\xi : Y \times V^+ \rightarrow 2^Y$ é a função de transição;
- $\psi : Y \rightarrow U$ é a função saída condição;
- y_0 é o estado inicial.

Para descrever os comportamentos lógicos dos SCEs são utilizadas linguagens em $(V^+ \times U)^*$ e para expressar o comportamento desejado para o sistema sob supervisão utiliza-se linguagens em V^*, ou seja, em termos de seqüências de eventos gerados pela planta. Assim, para que se possa relacionar o comportamento real com o comportamento desejado para o sistema em malha fechada, define-se a projeção $P_V : (V^* \times U) \rightarrow V^*$ como segue.

$$\begin{aligned} P_V(\epsilon) &= \epsilon \\ P_V(\sigma) &= \begin{cases} \epsilon, caso \ \sigma = \eta u \in (\{\eta\} \times U) \\ v, caso \ \ \sigma = vu \in (V \times U) \end{cases} \\ P_V(s\sigma) &= P_V(s)P_V(\sigma), caso \ s \in (V^+ \times U)^* \ e \ \sigma \in (V^+ \times U) \end{aligned} \tag{7.1}$$

Desta forma, a projeção $P_V : (V^+ \times U)^* \rightarrow V^*$ apaga os símbolos η e os símbolos $u \in U$ de palavras em $(V^+ \times U)^*$.

7.3.1.1 Abordagem por Controle de SEDs

O modelo de um SED com estrutura de controle é a tripla $P = (L, L_m, \Gamma)$, onde $L_m \subseteq L \subseteq (V^+ \times U)^*$ são, respectivamente, a linguagem marcada e gerada; e a estrutura de controle é um mapa $\Gamma : L \rightarrow 2^{2^{V^+ \times U}}$ tal que para todo $w \in L$ tem-se que $\Gamma(w) = \{\gamma \in 2^{V^+ \times U} : (\forall v \in V_L(w))(\exists u \in U_L(w, v)) : vu \in \gamma)\}$. Para compreender a estrutura de controle é necessário definir os seguintes conjuntos:

- $V_L(w)$ é o conjunto ativo de eventos em L após a cadeia $w \in \overline{L}$, e é definido por $V_L(w) = \{v \in V^+ : (\exists u \in U) w \cdot vu \in \overline{L}\}$;
- $U_L(w, v)$ é o conjunto ativo de condições em L após a cadeia $w \in \overline{L}$ para um dado evento $v \in V_L(w)$ e é definido com $U_L(w, v) = \{u \in U : w \cdot vu \in \overline{L}\}$.

A estrutura de controle depende da palavra gerada pelo sistema e traduz a idéia de que para um dado evento v, ativo após uma cadeia $w \in \overline{L}$, deve sempre haver

pelo menos uma condição u habilitada. Repare que a inibição de todas as condições possíveis para um dado evento levaria a uma situação sem contrapartida física, uma vez que o supervisor inibiria todas as condições aplicáveis em resposta a um dado evento.

Um supervisor S para o sistema $P = (L, L_m, \Gamma)$ é definido pelo mapa $S : L \rightarrow 2^{V^+ \times U}$. Um supervisor condição/evento equivalente à S pode ser definido como um SCE $\mathcal{S} : L \rightarrow V \times U$, tal que $(\forall w \in L)(\forall vu \in (V^+ \times U))w \cdot vu \in L \Leftrightarrow vu \in S(w)$. Ou seja, existe equivalência na ação de controle.

A linguagem $L(S/P) \subseteq L$ representa o comportamento da planta $P = (L, L_m, \Gamma)$ sob a ação do supervisor S e é definida recursivamente como:

1. $\epsilon \in L(S/P)$; e
2. $w \cdot vu \in L(S/P) \Leftrightarrow w \in L(S/P) \wedge w \cdot vu \in L \wedge vu \in S(w)$.

O comportamento marcado S/P é dado por $L_m(S/P) = L(S/P) \cap L_m$ e representa a parte da linguagem marcada da planta que sobrevive à ação de controle.

Dadas uma planta modelada como um SCE e especificações $A \subseteq E \subseteq V^*$, define-se o problema de controle supervisório equivalente para o modelo de SEDs $P = (L, L_m, \Gamma)$ com entrada de controle para a planta condição/evento como o de encontrar um supervisor S para P tal que $A \subseteq P_V[L_m(S/P)] \subseteq E$. Sendo $P_V(K) = E$ e $\emptyset \subset K \subseteq L_m$, existe um supervisor não-bloqueante S para P tal que $L_m(S/P) = K$ se e somente se K é L_m-fechada e vu-controlável em relação à L.

Sejam as linguagens $K \subseteq L \subseteq (V^+ \times U)^*$. K é dita ser vu-controlável em relação à L se $(\forall w \in \overline{K})V_L(w) = V_K(w)$. O conjunto de todas as linguagens de K que são vu-controláveis em relação à L, denotado por C_{VU} é não-vazio e fechado para união de conjuntos. Conseqüentemente, C_{VU} contém um elemento supremo único, chamado máxima linguagem vu-controlável e denotado por $supC_{VU}(K)$.

A solução do problema de controle supervisório equivalente requer, portanto, a realização dos seguintes passos:

1. Dada a especificação $E \subseteq V^*$, encontrar a linguagem-alvo $K \subseteq L_m \subseteq (V^+ \times U)^*$ tal que $P_V(K) = E$;
2. Verificar se K é vu-controlável em relação à L. Caso afirmativo, passar para o passo 3. Caso contrário, deve-se obter a máxima linguagem vu-controlável contida em K;
3. Implementar um autômato que represente o supervisor S tal que $L_m(S/P) = supC_{vu}(K)$.

7.3.1.2 Exemplo

Dadas uma planta modelada por um SCE e uma especificação $E \subseteq P_V(L_m)$ que representa o comportamento desejado para a planta em malha fechada, Figuras 7.4 e 7.5, respectivamente.

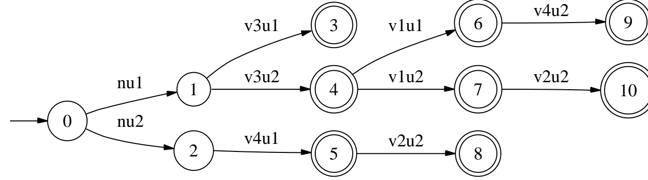


Figura 7.4: Planta Modelada por um SCE

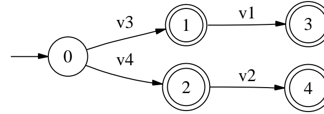


Figura 7.5: Especificação $E \subseteq P_V(L_m)$

O primeiro passo na síntese do supervisor consiste em obter a linguagem alvo $K \subseteq L_m \subseteq (V^+ \times U)^*$. Tal linguagem é obtida fazendo-se $K = P_V^{-1}(E) \cap L_m$ e o autômato que reconhece esta linguagem é ilustrado na Figura 7.6.

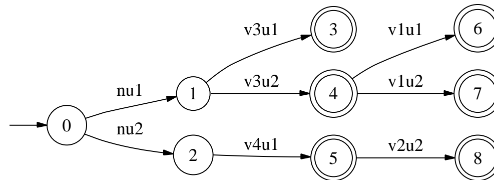


Figura 7.6: Especificação $K \subseteq L_m$

Pode-se verificar que a linguagem K não é vu-controlável em relação a L. Note, por exemplo, que após a cadeia $w = \eta u_1 \cdot v_3 u_2 \cdot v_1 u_1 \in \overline{K}$ pode ocorrer o evento v_4 na planta, mas o mesmo não está previsto na especificação, ou seja, $V_L(w) \nsubseteq V_K(w)$. De forma análoga, após a seqüência $w' = \eta u_1 \cdot v_3 u_2 \cdot v_1 u_2 \in \overline{K}$, a planta prevê a ocorrência do evento v_2, o qual não é previsto na especificação. Encontra-se então a máxima linguagem vu-controlável contida em K, denotada por $supC_{VU}(K)$, a qual é reconhecida pelo autômato apresentado na Figura 7.7.

Vale salientar que $L_m(S/P) = supC_{VU}(K)$. Desta forma, o supervisor pode ser implementado pelo autômato mostrado na Figura 7.7.

Por fim, pode-se obter a máxima linguagem v-controlável contida na especificação $E \subseteq V^*$ fazendo-se $supC_V(E) = P_V[supC_{VU}(K)]$. O autômato que reconhece esta linguagem é ilustrado na Figura 7.8.

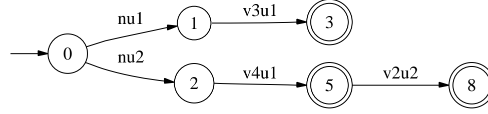


Figura 7.7: Máxima Linguagem vu-Controlável $supC_{VU}(K)$

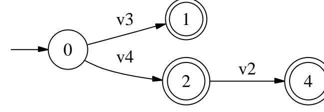


Figura 7.8: Máxima Linguagem v-Controlável $supC_V(E)$

7.3.2 Abordagem Modular

O problema de controle modular para SCEs consiste em desenvolver supervisores S_i, cada um para cumprir uma restrição de coordenação particular, e implementar o controle global através da conjunção dos supervisores S_i desenvolvidos.

Dadas uma planta modelada por um SCE e uma especificação $E \subseteq V^*$, tal que $E = E_1 \cap E_2$. Deseja-se encontrar supervisores condição/evento S_i tais que $P_V[L_m(S_i/P)] \subseteq E_i$, para $i = 1, 2$, e de forma que $S_1 \wedge S_2$ seja não-bloqueante para P e que $P_V[L_m((S_1 \wedge S_2)/P)] \subseteq E$.

7.3.2.1 Abordagem por Controle de SEDs

Diferentemente da teoria clássica de controle modular de SEDs, para SCEs não basta que as linguagens implementadas por supervisores modulares sejam não-conflitantes para garantir que a ação conjunta das mesmas seja não-bloqueante e ótima. Elas devem ser interconsistentes.

Duas linguagens $K_1, K_2 \subseteq (V^+ \times U)^*$ são ditas interconsistentes se $(\forall w \in \overline{K_1} \cap \overline{K_2}) V_{K_1}(w) \cap V_{K_2}(w) = V_{K_1 \cap K_2}(w)$. Em palavras, duas linguagens, K_1 e K_2, são interconsistentes se após qualquer cadeia comum a $\overline{K_1}$ e $\overline{K_2}$, os eventos possíveis de ocorrer em ambas as linguagens, são também possíveis de ocorrer na linguagem $K_1 \cap K_2$.

Se $supC_{VU}[P_V^{-1}(E_1) \cap L_m]$ e $supC_{VU}[P_V^{-1}(E_2) \cap L_m]$ são interconsistentes, então existem supervisores S_i não-bloqueantes para P que implementam as máximas linguagens v-controláveis contidas em E_i, e cuja ação conjunta dos supervisores resulta numa ação de controle não-bloqueante e ótima dada por $P_V[L_m((S_1 \wedge S_2)/P)] supC_V(E_1) \cap supC_V(E_2) = supC_V(E_1 \cap E_2)$. Ou seja, a solução obtida através da abordagem modular é idêntica àquela obtida utilizando-se a abordagem monolítica.

7.3.2.2 Exemplo

Dadas uma planta modelada por um SCE e as especificações $E_1, E_2 \subseteq P_V(L_m)$, Figuras 7.4 e 7.9, respectivamente.

Figura 7.9: Especificações $E_1, E_2 \subseteq P_V(L_m)$

O primeiro passo para a síntese de supervisores consiste em obter especificações equivalentes que estejam contidas na linguagem da planta. Tais especificações são obtidas fazendo-se $K_i = P_V^{-1}(E_i) \cap L_m$, para $i = 1, 2$. Os autômatos que reconhecem estas linguagens são ilustrados nas Figuras 7.10 e 7.11. Percebe-se que as linguagens K_1 e K_2 não são vu-controláveis em relação a L, e então obtém-se as máximas linguagens vu-controláveis contidas nestas especificações. Estas são reconhecidas pelos autômatos apresentados nas Figuras 7.12 e 7.13.

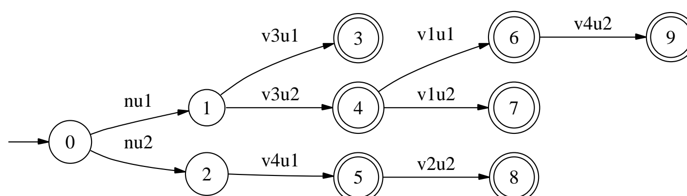


Figura 7.10: Especificação $K_1 \subseteq L_m$

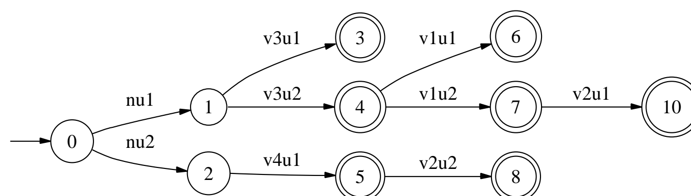


Figura 7.11: Especificação $K_2 \subseteq L_m$

Neste instante, testa-se a interconsistência entre $supC_{VU}(K_1)$ e $supC_{VU}(K_2)$. Neste caso tal condição não é verificada. Por isso, fixa-se uma destas linguagens (por exemplo, $supC_{VU}(K_2)$) e obtemos a máxima sublinguagem vu-controlável em relação a L e que é interconsistente em relação a linguagem fixada. Obtendo assim $supIC_{VU}(K_1^{\uparrow}, K_2^{\uparrow})$, onde $K_i^{\uparrow} = supC_{VU}(K_i)$, com $i = 1, 2$. Desta forma, os supervisores S_1 e S_2 podem ser implementados pelos autômatos mostrados nas Figuras 7.14 e 7.13, respectivamente.

Pode-se obter ainda a máxima linguagem v-controlável contida na especificação $E_i \subseteq V^*$ fazendo-se $supC_V(E_i) = P_V[supC_{VU}(K_i)]$ sendo $i = 1, 2$. Os autômatos que reconhecem estas linguagens são ilustrados na Figura 7.15. Por fim, a linguagem

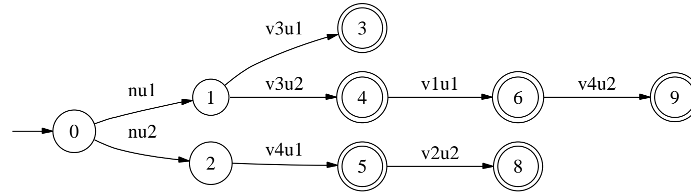


Figura 7.12: Máxima Linguagem vu-Controlável - $supC_{VU}(K_1)$

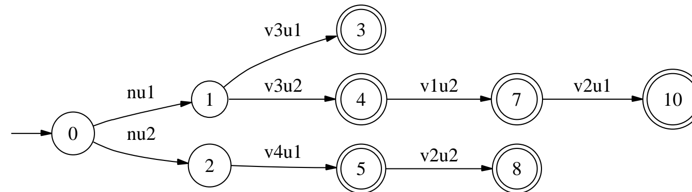


Figura 7.13: Máxima Linguagem vu-Controlável - $supC_{VU}(K_2)$

resultante da ação conjunta dos dois supervisores, dada por $P_V[L_m((S_1 \wedge S_2)/P)] = supC_V(E_1) \cap supC_V(E_2)$, é mostrada na Figura 7.16.

Comparando o resultado obtido na resolução deste problema com o resultado obtido através da abordagem monolítica, constata-se que $P_V[L_m((S_1 \wedge S_2)/P] = supC_V(E_1) \cap supC_V(E_2) = supC_V(E_1 \cap E_2)$, ou seja, a solução obtida através da abordagem modular é idêntica a obtida através da abordagem monolítica.

7.4 Grail - Módulo Condição/Evento

Esta seção descreve o módulo Condição/Evento, sendo este composto por uma classe derivada de `FM` cujos eventos são caracterizados por um par de inteiros. A criação deste módulo teve como objetivo a adaptação dos filtros implementados em [Rodrigues, 2004] à nova estrutura do Grail.

7.4.1 Classes

No Grail, um SCE é representado por uma tripla $IOS = (Q_i, \Theta, Q_f)$, onde Q_i é o conjunto de estados iniciais, Θ é o conjunto de instruções e Q_f é o conjunto de estados finais. Cada instrução final $\theta \in \Theta$ é composta por:

- Um estado fonte θ_{source}.
- Uma etiqueta de transição θ_{event}, representado por um par de inteiros. O primeiro inteiro indica o evento e o segundo a condição.

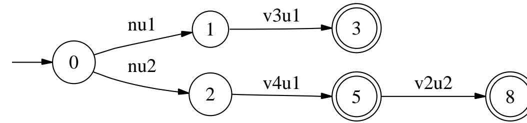


Figura 7.14: $SupIC_{VU}(K_1^{\uparrow}, K_2^{\uparrow})$

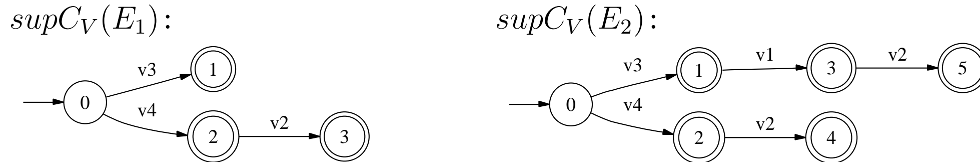


Figura 7.15: Máximas Linguagens v-Controláveis

- Um estado destino θ_{sink}.

Um exemplo é mostrado na Figura 7.17.

A classe que define o IOS dentro do Grail para Controle Supervisório denomina-se iots. Esta trata-se de uma instanciação de fm pela classe pair, ou seja, iots é uma fm<pair<int,int>>.

7.4.2 Filtros

Os filtros pertencentes à classe iots são:

1. ioinputs - retorna um autômato cuja linguagem é idêntica à do autômato dado, sendo essa, entretanto, composta apenas por eventos.
   - **Sintaxe:** ioinputs io1
2. ioisinter - testa a interconsistência entre as linguagens de dois autômatos distintos e passados como parâmetro.
   - **Sintaxe:** ioisinter io1 io2
3. iok - obtém a linguagem-alvo $K \subseteq (V^+ \times U)^*$ equivalente à uma especificação $E \subseteq V^*$ passada como parâmetro. É necessário também a passagem da planta P como parâmetro.
   - **Sintaxe:** iok P E
4. ioproj - obtém de forma sistemática a projeção da linguagem em $(V^+ \times U)^*$ do autômato de entrada em uma linguagem contida em V^*.
   - **Sintaxe:** ioproj io1

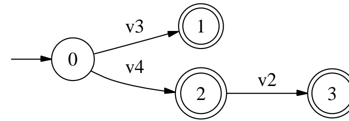
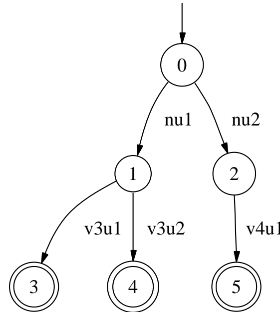


Figura 7.16: $P_V[L_m((S_1 \wedge S_2)/P)] = supC_V(E_1) \cap supC_V(E_2)$

SCE:



IOS:

```
(START) |- 0
0 [0,1] 1
0 [0,2] 2
1 [3,1] 3
1 [3,2] 4
2 [4,1] 5
3 -| (FINAL)
4 -| (FINAL)
5 -| (FINAL)
```

Figura 7.17: Representação de um SCE por um IOS

5. `iosupc` - computa a máxima linguagem de K que é vu-controlável em relação à planta P.
   - **Sintaxe:** `iosupc P K`
6. `iosupi` - obtém de forma sistemática a máxima linguagem de K_1 que é interconsistente em relação a K_2.
   - **Sintaxe:** `iosupi K1 K2`
7. `iosupci` - computa a máxima linguagem de K_1 que é vu-controlável em relação à planta P e interconsistente em relação a máxima linguagem de K_2 que é vu-controlável.
   - **Sintaxe:** `iosupci P K1 K2`

7.5 Conclusões

Este capítulo teve como objetivo a familiarização do leitor com o módulo Condição/Evento e seus filtros. Como perspectiva, avalia-se a possibilidade da adição dos algoritmos da operação empilhar e da encontrar o autômato condição/evento desenvolvidos em [Garcia, 2002]. A adição destes filtros ao Grail para Controle Supervisório tornaria este ambiente auto-suficiente na resolução de problemas que envolvessem Sistemas Condição/Evento.

Capítulo 8

Conclusões e Perspectivas

O Grail é um ambiente de computação simbólica que envolve linguagens finitas, expressões regulares e máquinas de estados finitos. Por se tratar de um programa de código aberto, o Grail versão 2.5 foi sendo expandido, de forma descontrolada, com o objetivo de focar sua utilização na Teoria de Controle Supervisório. Uma das contribuições deste trabalho foi a reorganização e a definição de uma estrutura mais adequada à expansão do mesmo, sendo o resultado disso o denominado Grail para Controle Supervisório.

O Grail para Controle Supervisório é um ambiente computacional que auxilia o desenvolvimento da Teoria de Controle Supervisório. Em paralelo com a estruturação do Grail para Controle Supervisório, novas funcionalidades foram adicionadas ao mesmo. Cita-se aqui a implementação dos filtros de verificação de L-fechamento, controlabilidade e não-conflito entre supervisores modulares. Destaca-se a implementação do filtro de redução de supervisores, cujo algoritmo é polinomial mesmo computando o supervisor reduzido através de coberturas de controle. Fica como perspectiva a criação de novos filtros, cujo procedimento de criação deve ser o especificado no Capítulo 3.

Como a Teoria de Controle Supervisório atua como base para vários ramos de expansão que atuam sobre problemas diferenciados, o Grail para Controle Supervisório foi estruturado de forma a também ser uma base cujos ramos são definidos por seus vários módulos. No decorrer deste trabalho, a estrutura dos possíveis módulos foi definida e três deles foram desenvolvidos, o Condição/Evento, o Hierárquico e o Multitarefa. Fica como perspectiva a criação de módulos que venham a lidar com outros ramos da TCS, como por exemplo os SEDs modelados por sistemas de transição de estados e estrutura de dados [de Oliveira, 2005], e a expansão dos módulos existentes através da criação de novos filtros.

Por ser um ambiente voltado à facilidade de expansão, o Grail apresenta certas limitações quanto à velocidade de processamento de seus algoritmos e quanto ao número

máximo de estados e transições das máquinas de estados finitos. O Grail perde muito tempo na criação e destruição de objetos devido à alocação dinâmica de memória. No Grail versão 2.5 existe uma classe denominada `pool`, cujo único objetivo é eliminação deste tempo desperdiçado. Entretanto, esta classe foi adaptada apenas para REs e uma extensão desta classe para FMs fica como perspectiva para futuros trabalhos.

Por último, foram desenvolvidos, no decorrer deste trabalho, filtros de conversão de formatos. Estes facilitam ao usuário a utilização de outros programas, como o TCT, para a utilização de funcionalidades que não tenham sido implementadas no Grail.

Bibliografia

Bouzon, G., de Oliveira, M., Vallim, M., Lacombe, J., Freitas, G., Cury, J. E., and Farines, J.-M. (2004). Cebe: Uma plataforma para experimentação real aplicada ao ensino de sistemas a eventos discretos. *Congresso Brasileiro de Automática.*

Cury, J. E. R. (2001). Teoria de controle supervisório de sistemas a eventos discretos. `http://www.das.ufsc.br/~cury/ensino-5202.html`.

Cury, J. E. R. (2005). José eduardo ribeiro cury. `http://www.das.ufsc.br/~cury`.

Cury, J. E. R., Torrico, C. R. C., and da Cunha, A. E. C. (2004). Supervisory control of discrete event systems with flexible marking. *European Journal of Control*, 10:47–60.

da Cunha, A. E. C. (2003). *Contribuições ao Controle Hierárquico de Sistemas a Eventos Discretos.* Tese de doutorado, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis.

de Oliveira, C. (2005). *Contribuições ao Problema de Controle Supervisório de Sistemas a Eventos Discretos Parametrizáveis e Não-Regulares.* Tese de doutorado, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis.

de Queiroz, M. H. (2000). *Controle Supervisório Modular de Sistemas de Grande Porte.* Dissertação de mestrado, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis.

de Queiroz, M. H. (2004). *Controle Supervisório Modular e Multitarefa de Sistemas Compostos.* Tese de doutorado, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis.

de Queiroz, M. H., Cury, J. E. R., and Wonham, W. (2005). Multitasking supervisory control of discrete event systems. *A ser publicado no Journal of Discrete Event Dynamic Systems*, 15(4).

Deitel, H. and Deitel, P. (2001). *C++ Como Programar.* Bookman, Porto Alegre, 3 edition. Trad. Lisbôa, C. and Lisbôa, M.

Fowler, M. (2003). *UML Distilled - A Brief Guide to the Standard Obeject Modeling Language*. Addison-Wesley, 3 edition.

Garcia, T. (2002). *Controle Supervisório de Sistemas a Eventos Discretos: Uma Abordagem por Modelo Condição/Evento*. Dissertação de mestrado, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis.

Leal, A. B. (2005). *Controle Supervisório Modular de Sistemas Híbridos*. Tese de doutorado, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis.

Moore, E. (1964). *Sequential Machines Selected Papers*. Addison-Wesley, USA.

Ramadge, P. and Wonham, W. (1989). The control of discrete event system. *Proceedings of the IEEE*, 77(1).

Raymond, D. (2005). Darrell raymond's home page. `http://db.uwaterloo.ca/~drraymon/`.

Raymond, D. and Wood, D. (1996a). Grail: Engineering automata in c++: Version 2.5. `http://www.csd.uwo.ca/research/grail/.papers/engine.ps`.

Raymond, D. and Wood, D. (1996b). Programmer's guide to grail: Version 2.5. `http://www.csd.uwo.ca/research/grail/.papers/prog.ps`.

Raymond, D. and Wood, D. (1996c). Release notes for grail: Version 2.5. `http://www.csd.uwo.ca/research/grail/.papers/notes.ps`.

Raymond, D. and Wood, D. (1996d). User's guide to grail: Version 2.5. `http://www.csd.uwo.ca/research/grail/.papers/user.ps`.

Raymond, D. and Wood, D. (2002). Grail. `http://www.csd.uwo.ca/research/grail/`.

Rodrigues, D. L. (2004). *Implementação de Algoritmos para o Controle Supervisório Modular de Sistemas Condição/Evento*. Dissertação de mestrado, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis.

Stroustrup, B. (2000). *The C++ Programming Language*. Addison-Wesley, 4 edition.

Su, R. and Wonham, W. (2003). Supervisor reduction for discrete-event systems.

Torrico, C. R. C. (1999). *Implementação de Controle Supervisório de Sistemas a Eventos Discretos Aplicado a Processos de Manufatura*. Dissertação de mestrado, Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianópolis.

UMDES (2005). University of michigan - discrete event system group. `http://www.eecs.umich.edu/umdes/toolboxes.html`.

Vaz, A. and Wonham, W. (1986). On supervisor reduction in discrete-event systems. *International Journal of Control*, 44(2):475–491.

Wonham, W. (2005). W.m.wonham's home page. `http://www.control.toronto.edu/people/profs/wonham/wonham.html`.

Wood, D. (2005). Derick wood (ph.d, leeds, 1968). `http://www.cs.ust.hk/faculty/dwood/`.