Yoshiaki Date

Uma proposta de arquitetura de sistemas de controle utilizando padrões de projeto e CSP-OZ

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

CONSULTA
FD-4336
Ed.rev.

São Paulo
2006

OK

Yoshiaki Date

Uma proposta de arquitetura de sistemas de controle utilizando padrões de projeto e CSP-OZ

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Engenharia.

Área de concentração:
Engenharia de Controle e Automação Mecânica - Engenharia Mecatrônica

Orientador:
Prof. Dr. Newton Maruyama

São Paulo
2006

À toda minha família.

Agradecimentos

Meus sinceros agradecimentos ao meu orientador, Newton Maruyama, pela sua forma paciente, apoio e dedicação que me orientou.

À minha família, pela dedicação, pelo carinho e pela paciência.

À minha namorada, pelo apoio e pela dedicação.

Aos meus amigos, que me apoiaram e incentivaram durante todo o decorrer do curso.

E meus sinceros agradecimentos a todos aqueles que se envolveram no processo de desenvolvimento deste trabalho.

Resumo

O desenvolvimento de software de sistemas de controle pode ser caracterizado como um trabalho de alta complexidade. A sua alta complexidade se deve principalmente às características de não determinismo, assincronismo e concorrência entre os seus processos. Este trabalho propõe uma nova arquitetura de sistemas de controle baseada em padrões de projeto e na utilização da combinação de linguagens de especificações formais CSP e *Object-Z* (CSP-OZ). A linguagem *Object-Z* é utilizada para se especificar características estáticas e métodos de classes através da utilização de predicados, enquanto que CSP é uma álgebra de processos utilizada para especificar a dinâmica de processos concorrentes. A proposta desta arquitetura visa a construção de uma arquitetura robusta, correta, de fácil modificação, reutilizável e que possa servir de base para implementação de projetos futuros.

A arquitetura do software é dividida em duas partes. Na primeira parte, alguns padrões de projeto comumente utilizados em sistemas de controle são especificados utilizando-se a linguagem de especificação formal CSP-OZ. As especificações formais são convertidas em programas JAVA através da utilização da biblioteca JCSP (uma biblioteca construída para traduzir facilmente especificações CSP). Na segunda parte, as aplicações utilizando CSP-OZ e JCSP são construídas baseadas em uma biblioteca de padrões de projeto que foram previamente analisadas. Um estudo de caso é utilizado para se testar e validar a arquitetura proposta.

Abstract

The development of control system software might be characterized as a highly complex task. This is mainly due to the characteristics of nondeterminism, asynchronicity and concurrency between processes. In this work, a new control system architecture based on design patterns and the use of a combination of formal specification languages CSP and Object-Z (CSP-OZ) is proposed. Object-Z can specify the state space and methods of a class using predicates while the CSP is a process algebra that can specify the dynamics of concurrent processes. The proposal aims at designing a robust, provable, maintainable and reusable architecture that can serve as a base of future project implementations.

The software architecture is designed in two parts. In the first part, some design patterns which are commonly used for control systems are specified using CSP-OZ formal specification language. These formal specifications are converted into JAVA programs using the JCSP library, a software library designed to easily translate CSP specifications. In the second part, applications using CSP-OZ and JCSP might be designed based on the library of design patterns that are previously provable correct. A case study is used to test and validate the proposed architecture.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução **1**

1.1 Motivação 2

1.2 Objetivos 3

1.3 Exemplificação 4

1.4 Organização do trabalho 8

2 Especificações formais **10**

2.1 Especificações formais 11

2.1.1 Combinações de especificações formais 13

2.2 Object Z 15

2.2.1 Visibilidade dos objetos 17

2.2.2 Agregação 18

2.2.3 Composição 19

2.2.4 Herança 21

2.2.5 Variáveis secundárias 22

2.2.6 Polimorfismo 23

2.2.7 *Containment* 24

2.3 CSP 25

2.3.1 Sintaxe CSP 27

2.3.2 Controle de Fluxo 31

2.3.3 *Traces* 33

2.3.4 Especificação e validação com *traces* 36

2.3.5 *Stable Failures* 36

2.3.6 *Acceptances* 37

2.4 Combinando *Object-Z* com CSP 38

2.4.1 *Failures* e *Divergencies* 39

2.4.2 Objetos ativos e objetos passivos 41

2.4.3 Comunicação externa com o ambiente 41

2.5 Exemplo: Produtor e Consumidor 42

2.5.1 Descrição do exemplo 43

2.5.2 Biblioteca JCSP 43

2.5.3 Especificação 44

3 Padrões de projeto **50**

3.1 Introdução 50

3.1.1 Importância na utilização de padrões de projeto 51

3.1.2 Classificação de padrões de projeto 52

3.1.3 Critérios de escolha de padrões de projeto 53

3.2 Padrões de projeto para sistema de tempo real 55

3.2.1 Padrões de Distribuição de Recursos 55

3.2.2 Padrão *Data Bus* - Descrição 56

3.2.3 Escalonamento de Processos 65

3.2.4 Padrões de Concorrência - Padrão de prioridade estática . 65

3.2.5 Processos de leitura e escrita de dados 76

3.2.6 Padrões de Segurança 77

3.2.7 Implementação de um *WatchDog* 78

3.2.8 Tratamento de Eventos Assíncronos 80

4 Estudo de caso **82**

4.1 Introdução 82

4.1.1 Descrição do sistema 82

4.2 Processos que compõem o estudo de caso 86

4.2.1 Classes responsáveis pelo carregamento da cana-de-açúcar 89

4.2.2 Especificação de Eventos Assíncronos 96

4.2.3 Classe principal 99

4.3 Validação por *traces* 101

4.3.1 Refinamento da classe *GuindasteProcess* 101

4.3.2 Refinamento da classe *CaminhaoCacambaProcess* 101

4.3.3 Refinamento da classe *CaminhaoCorrentesProcess* 102

4.3.4 Refinamento da classe *Mesa1Process* 102

4.3.5 Refinamento da classe *TransporteComGarrasProcess* . . . 102

4.3.6 Refinamento da classe *Mesa2Process* 102

4.3.7 Refinamento da classe *FossoProcess* 102

5 Implementação e Simulação **103**

5.1 Estrutura 104

5.2 Considerações particulares 108

5.2.1 Implementação de atuadores e sensores 108

5.2.2 Implementação de prioridades 109

5.2.3 Implementação de Eventos Assíncronos 110

5.3 Experimentos Realizados 111

5.3.1 Leitura de dados 112

5.3.2 Leitura de eventos assíncronos 114

5.3.3 Recepção de cana-de-açúcar 115

6 Conclusão **116**

6.1 Principais contribuições 117

6.2 Trabalhos Futuros 118

Referências **119**

Apêndice A - Diagramas do estudo de caso **124**

A.1 Diagramas de sensores 124

A.2 Diagramas de atuadores 126

Lista de Figuras

1.1 Esquematização do trabalho. 3
1.2 Arquitetura do sistema. 4
1.3 Exemplo de um sistema com dois robôs. 5
2.1 Diagrama de classes do sistema produtor-consumidor. 49
3.1 Diagrama de classes do padrão de projeto *DataBus*. 57
3.2 Diagrama de classes de escalonamento de processos. 67
3.3 Diagrama entre os processos. 72
3.4 Diagrama de estados. 75
4.1 Recepção de cana. 84
4.2 Recepção e lavagem de cana. 85
5.1 Diagrama de componentes. 104
5.2 Dados de entrada e saída. 105
5.3 Processos básicos do sistema. 106
5.4 Processos do sistema de carregamento de cana-de-açúcar. . . . 107
5.5 Painel de eventos assíncronos. 114

Lista de Tabelas

4.1 Classes relacionadas à leitura de dados do sistema. 87

4.2 Classes relacionadas à escrita de dados do sistema. 88

5.1 Tabela com dados de sensores simulados. 109

5.2 Taxa de leitura dos sensores. 114

1 Introdução

O aumento crescente da complexidade dos sistemas computacionais nos últimos anos, e a demanda cada vez maior por qualidade e confiabilidade destes sistemas têm aumentado consideravelmente a importância de pesquisas relacionadas à melhoria de qualidade de software, através do estudo de novas metodologias e técnicas no desenvolvimento de software (KELLY; MCDERMID, 2004; WILSON; KELLY; MCDERMID, 1997).

Em relação ao estudo de sistemas de tempo real, a sua complexidade é acentuada, principalmente devido a sua natureza não determinística, a existência de assincronismos entre os processos que a compõem e a variação no tempo, bastante comum a estes sistemas. Sistemas de tempo real são sistemas aonde a previsibilidade de tempo de execução de um processo é essencial para o funcionamento correto do mesmo. Um requisito de tempo pode ser, por exemplo, a quantidade de tempo que um sistema crítico pode tolerar antes da ocorrência de um incidente devido às características de estabilidade de um controlador PID (TORNGREN, 1998; BATE; CERVIN; NIGHTINGALE, 2003). Um dos modos mais comuns de se modelar tempo para estes sistemas é definindo um intervalo de tempo, começando com a ocorrência de um evento ou o início de execução de um processo e o fim de execução deste processo, chamado de *deadline*. Sistemas podem ser analisados através da sua escalabilidade, periodicidade, prioridade, o pior tempo de execução e o *deadline* de todos os processos.

A correção de falhas em tais sistemas muitas vezes é dificultada pela impossibilidade de se reproduzir cenários de testes e falhas, e conseqüentemente o seu estudo e sua especificação apresentam uma complexidade mais acentuada em relação a sistemas não concorrentes (KELLY; MCDERMID, 2004; WILSON; KELLY; MCDERMID, 1997).

Neste capítulo introdutório, é apresentada a motivação principal para se especificar sistemas concorrentes, sob o ponto de vista formal, os objetivos propostos por este trabalho e a estrutura geral desta dissertação.

1.1 Motivação

O desenvolvimento de sistemas computacionais envolve o uso de modelos abstratos e precisos que permitem a uma equipe de engenheiros de software especificar, projetar, implementar e manter sistemas de software de modo a garantir os requisitos de qualidade pré-estabelecidos. Além destes aspectos técnicos, a engenharia de software envolve mecanismos para se planejar e gerenciar os processos de desenvolvimento de software, bem como as interações existentes entre as diversas pessoas que compõem uma equipe de desenvolvimento de software.

Sob o ponto de vista do produto, ou seja, os aspectos técnicos utilizados para se desenvolver o software, esforços consideráveis têm sido gastos para se conseguir desenvolver softwares livres de erros e aderentes aos requisitos definidos logo nos estágios iniciais de desenvolvimento, nos quais a identificação de erros tem um impacto menor e custos menores no desenvolvimento de sistemas. Podemos destacar, segundo Hoare (2003), as seguintes melhorias técnicas:

- Compiladores otimizados indicando, por exemplo, possíveis erros em tempo de compilação,
- Linguagens e métodos de especificação de sistemas,
- Novas teorias de programação,
- Utilização de padrões de projeto,
- Utilização de *assertions* e testadores automáticos, tornando o processo de análise e detecção de erros de maneira automática, permitindo-se desta maneira, a sua correção,
- Ferramentas automáticas, como checadores de modelos e provadores de teoremas.

Em relação às linguagens e aos métodos de especificação de sistemas, diversas propostas têm sido utilizadas e elaboradas, tanto na área acadêmica como industrial, tendo um impacto grande na qualidade do software. Entende-se por especificações de sistemas a definição das funcionalidades que um sistema computacional deve executar. Sua especificação pode se basear em uma abordagem informal, por exemplo, um manual ou documentos descrevendo o desenvolvimento do sistema, ou uma abordagem formal, na qual a sua definição é estruturada em um sistema correto e completo, podendo desta maneira, validar as propriedades de sua especificação de uma maneira sistemática.

A utilização de métodos formais permite a verificação das propriedades do software e hardware de modo a garantir que o sistema funcione dentro das características pré-definidas. A verificação de programas não significa que um programa seja livre de erros, uma vez que o procedimento de decisão para um sistema axiomático genérico não existe (usa-se de heurísticas para que o provador de teoremas avance a algum resultado final), porém, podem reduzir consideravelmente a quantidade de testes necessários. Atualmente existem diversas ferramentas baseadas em provadores automáticos de teoremas e "checadores"de modelos em desenvolvimento, utilizadas para a verificação de sistemas.

Em relação a sistemas computacionais críticos, onde a sua falha pode acarretar perdas financeiras, perda de credibilidade, perda de vidas humanas ou acidentes ambientais, a sua confiabilidade e segurança são requisitos primordiais, portanto, a utilização de métodos formais pode colaborar para o desenvolvimento e uma melhor compreensão de sistemas de alta integridade.

1.2 Objetivos

O objetivo deste trabalho é propor um modelo de uma arquitetura de um sistema geral de controle, baseado na utilização de métodos formais aplicados a padrões de projeto mais utilizados na implementação de sistemas de tempo real, de modo a torná-lo flexível e reutilizável, e que através de uma adaptação adequada, a sua aplicabilidade se estenda a uma grande quantidade de sistemas.

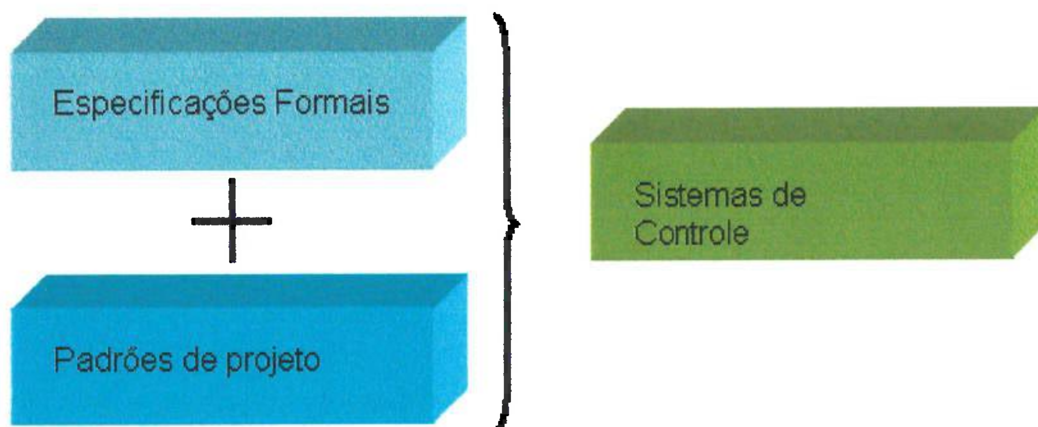


Figura 1.1: Esquematização do trabalho.

Para se atingir tal objetivo, o trabalho pode ser separado nas seguintes etapas, visualizadas na figura 1.1:

1. Implementação dos padrões de projeto mais utilizados para o desenvolvimento de um sistema de tempo real, escolhendo as características comuns

entre os diversos padrões e que apresente um maior grau de flexibilidade.

2. Analisar através da utilização de métodos formais as características estáticas e dinâmicas que compõem cada um dos subsistemas, bem como as interações existentes entre elas.

3. Definir uma arquitetura lógica que compõe o sistema e suas principais considerações. Apresentar uma biblioteca básica para futuras aplicações e indicar as alterações necessárias para adaptá-los de maneira adequada.

Com a definição deste modelo de arquitetura comum, através da utilização de métodos formais, este trabalho pretende contribuir para a melhoria da qualidade no desenvolvimento de um sistema de tempo real, bem como diminuir o ciclo de desenvolvimento, uma vez que a solução do problema parte de uma arquitetura previamente analisada e implementada.

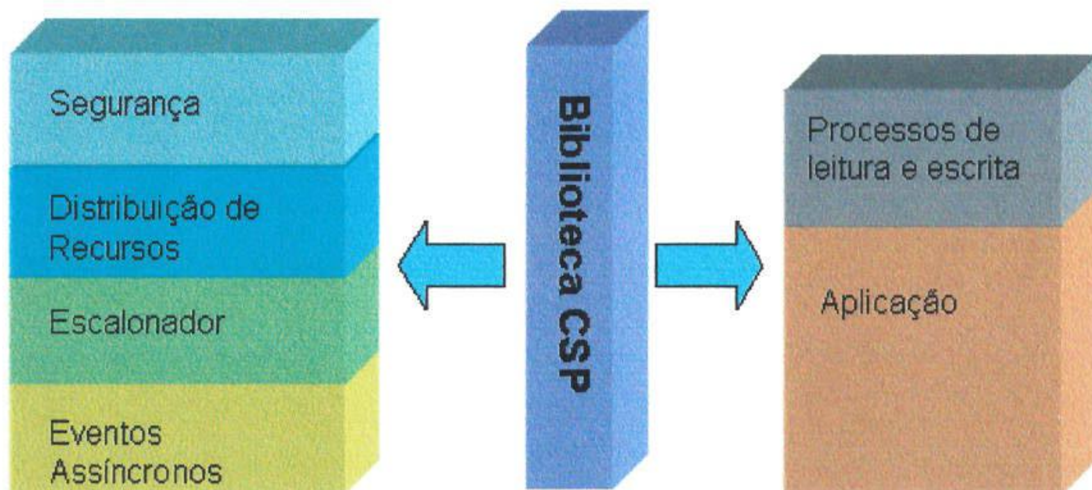


Figura 1.2: Arquitetura do sistema.

A arquitetura proposta pode ser visualizada na figura 1.2, na qual os padrões de projeto são referentes a cada um dos blocos estruturados no lado esquerdo da figura, no caso, os padrões de segurança, distribuição de recursos, escalonamento e tratamento de eventos assíncronos. A biblioteca CSP (uma implementação dos operadores da linguagem de especificação formal CSP para uma linguagem de programação, cuja descrição se encontra no capítulo 2) serve de ligação entre a especificação formal e os padrões de projeto.

1.3 Exemplificação

Um exemplo da utilização desta arquitetura pode ser visualizado através de uma situação em que um robô retira peças de um *buffer* e a coloca em uma estação de

trabalho, um segundo robô executa as operações necessárias e a transfere para um segundo *buffer*. A esquematização deste pequeno exemplo é ilustrada na figura 1.3.

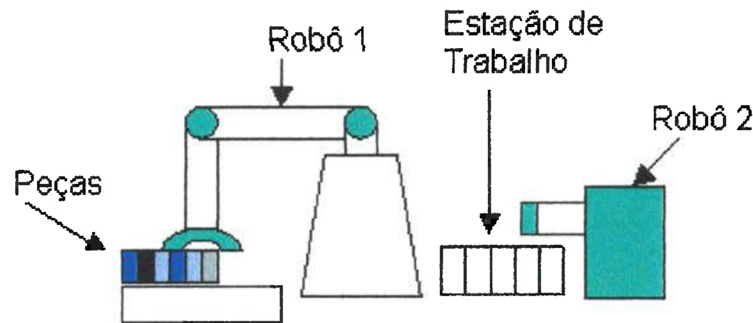


Figura 1.3: Exemplo de um sistema com dois robôs.

Os dados de entrada e saída que compõem o processo de leitura e escrita esquematizado na figura 1.2 deste pequeno exemplo são listados logo abaixo:

1. A existência de peças no *buffer* número 1.
2. A existência de peça na estação de trabalho.

Cada um destes dados de entrada e saída é atualizado pelo processo de leitura e escrita de dados, ilustrado na figura 1.2 e que por sua vez, atualiza os valores gerenciados pelo padrão de projeto relacionado à distribuição de recursos (descritos no capítulo 3), o restante da aplicação utiliza os dados uma vez armazenados ao invés de acessá-los diretamente.

Os processos que compõem o sistema e são relacionados à aplicação são listados logo abaixo:

1. O robô número 1 verifica a existência de peças no *buffer* 1 e se a estação de trabalho está livre. Caso as duas condições sejam satisfeitas, o robô número 1 envia a peça do *buffer* à estação de trabalho.
2. O robô número 2 verifica a existência de peças na estação de trabalho e, caso exista, processa e envia a peça ao *buffer* número 2.

O processo descrito acima é especificado através do método formal CSP-OZ adotado por este trabalho. Uma introdução teórica a respeito de CSP-OZ é descrita no capítulo subseqüente. Este pequeno exemplo utiliza-se dos padrões de projeto responsável pela distribuição de recursos descritos no capítulo 3.

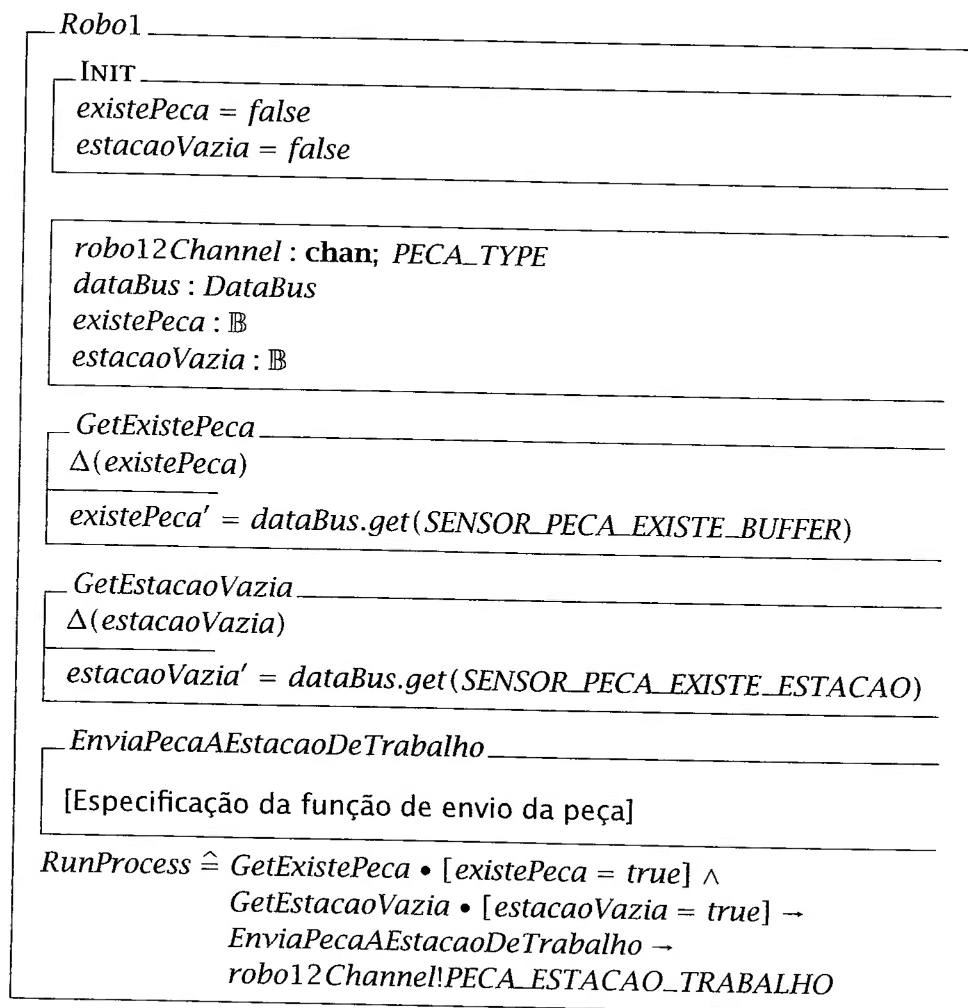
Abaixo, a definição dos tipos utilizados neste exemplo, o tipo *PECA_TYPE* é utilizado pelo canal de sincronização CSP entre os dois robôs, enquanto que o tipo

ID_TYPE é utilizado para se retornar os valores dos dados armazenados pelo sistema.

$$PECA_TYPE ::= PECA_ESTACAO_TRABALHO$$

$$ID_TYPE ::= SENSOR_PECA_EXISTE_BUFFER \mid SENSOR_PECA_EXISTE_ESTACAO$$

Abaixo, a especificação do Robô número 1. Os métodos de envio da peça a estação de trabalho não foram especificadas e podemos observar que os valores dos sensores são obtidos indiretamente através da utilização da classe *DataBus*, ou seja, os dados de entrada e saída do sistema são centralizados em uma única classe, facilitando-se desta maneira a sua análise e implementação.



Robo1

INIT
$existePeca = false$
$estacaoVazia = false$

$robo12Channel : \textbf{chan}; PECA_TYPE$
$dataBus : DataBus$
$existePeca : \mathbb{B}$
$estacaoVazia : \mathbb{B}$

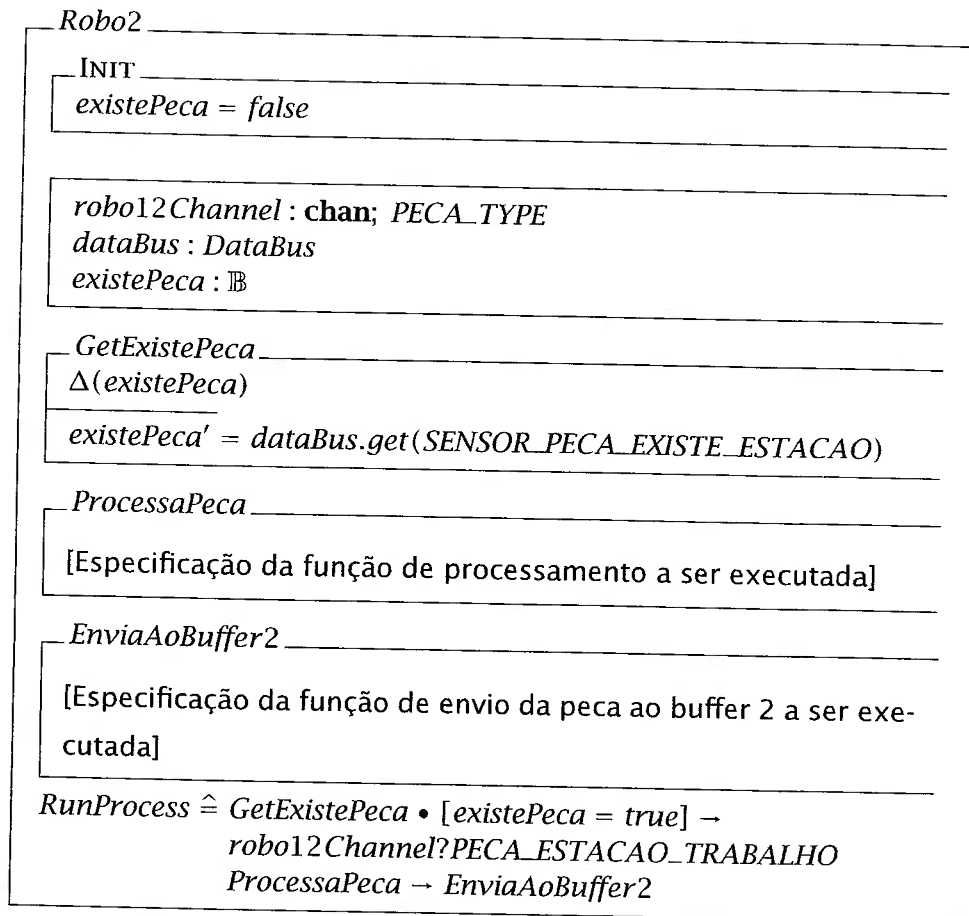
GetExistePeca
$\Delta(existePeca)$
$existePeca' = dataBus.get(SENSOR_PECA_EXISTE_BUFFER)$

GetEstacaoVazia
$\Delta(estacaoVazia)$
$estacaoVazia' = dataBus.get(SENSOR_PECA_EXISTE_ESTACAO)$

EnviaPecaAEstacaoDeTrabalho
[Especificação da função de envio da peça]

$RunProcess \mathrel{\widehat=} GetExistePeca \bullet [existePeca = true] \wedge$
$GetEstacaoVazia \bullet [estacaoVazia = true] \rightarrow$
$EnviaPecaAEstacaoDeTrabalho \rightarrow$
$robo12Channel!PECA_ESTACAO_TRABALHO$

O robô número 2 é sincronizado com o robô número 1 através do canal CSP *robo12Channel*, a escrita no canal é efetuada pelo robô número 1 e a sua leitura pelo robô número 2. A utilização destes canais é proveniente da especificação formal CSP, descrito no capítulo 2, em que os processos ficam em estado de espera até que a comunicação entre eles esteja habilitada.



Robo2

INIT
existePeca = *false*

robo12Channel : **chan**; *PECA_TYPE*
dataBus : *DataBus*
existePeca : $\mathbb{B}$

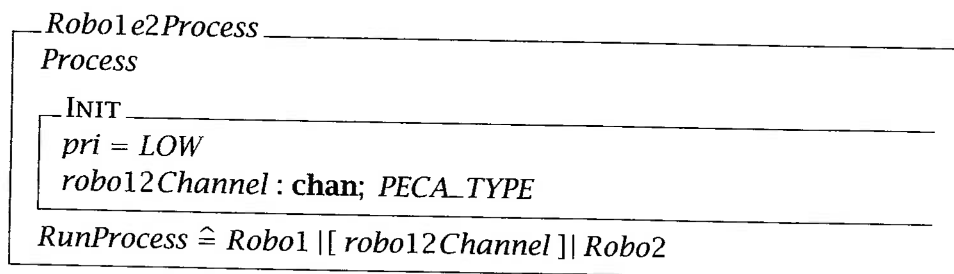
GetExistePeca
Δ(*existePeca*)
existePeca' = *dataBus.get*(*SENSOR_PECA_EXISTE_ESTACAO*)

ProcessaPeca
[Especificação da função de processamento a ser executada]

EnviaAoBuffer2
[Especificação da função de envio da peca ao buffer 2 a ser executada]

RunProcess $\widehat{=}$ *GetExistePeca* • [*existePeca* = *true*] →
robo12Channel?*PECA_ESTACAO_TRABALHO*
ProcessaPeca → *EnviaAoBuffer2*

Abaixo, o processo relacionado ao robô número 1 e ao robô número 2. É possível observarmos que os processos *Robo1* e *Robo2* são sincronizados através do canal *robo12Channel*, como descrito anteriormente. A classe *Robo1e2Process* é derivada da classe *Process*, na qual são definidas as características básicas de um processo utilizadas pela classe responsável pelo seu escalonamento como, por exemplo, a prioridade e a freqüência do disparo do seu processamento.



Robo1e2Process
Process

INIT
pri = *LOW*
robo12Channel : **chan**; *PECA_TYPE*

RunProcess $\widehat{=}$ *Robo1* |[*robo12Channel*]| *Robo2*

E por fim, cada um dos processos listados acima é escalonado de acordo com a sua prioridade, o seu controle de execução é feito através do padrão de projeto responsável pelo escalonamento de processos. Vale ressaltar que o processo *Robo1e2Process* executa em paralelo com o processo de leitura de dados e o processo responsável pela monitoração de segurança. A especificação do sistema como um todo é descrito abaixo:

$$Main \mathrel{\widehat=} scheduler \parallel DataListener \parallel$$
$$Robo1e2Process \parallel WatchDogProcess$$

Onde o sistema é especificado utilizando-se o operador de paralelismo CSP ||, sincronizando-se os processos *scheduler*, *DataListener*, *Robo1e2Process* e *WatchDogProcess*.

1.4 Organização do trabalho

O capítulo 2 introduz os métodos formais e sua utilização, bem como às qualidades desejáveis de um método formal. Neste capítulo são descritos conceitos de *Object-Z*, um método formal orientado a objetos, derivado da especificação Z (SPIVEY, 1992), uma introdução à CSP, um método formal, baseado na dinâmica e no comportamento de sistemas concorrentes e paralelos, descrevendo as suas principais características, seus operadores e as diferenças entre as suas versões, bem como as evoluções descritas inicialmente por C. A. R. Hoare (HOARE, 1985), Roscoe (ROSCOE, 1998), Schneider (SCHNEIDER, 2000) e Lawrence (LAWRENCE, 2003b, 2002, 2003a, 2004), e por fim, a combinação de *Object-Z* e CSP é descrita brevemente neste capítulo, sendo que a primeira é responsável pelas características estáticas do sistema e a segunda pelas características dinâmicas e a descrição da semântica adotada pela combinação. No final do capítulo é descrito um exemplo de um sistema produtor, consumidor especificado através da utilização da linguagem de especificação CSP-OZ e os passos de refinamento até a sua transformação para uma linguagem de programação.

O capítulo 3 é uma introdução aos padrões de projeto, descrevendo a sua importância atual no desenvolvimento de sistemas baseado em orientação a objetos, sendo que a ênfase dada a estes padrões são referentes a sistemas de tempo real. Uma análise dos padrões de projetos é efetuado através da combinação de *Object-Z* e CSP neste capítulo. Dentre os padrões abordados, podemos destacar o escalonamento de processos, alocação de recursos, padrões de segurança e leitura e escrita de dados importantes ao sistema.

No capítulo 4, os mesmos padrões de projetos analisados no capítulo anterior são aplicados a um estudo de caso, demonstrando a flexibilidade existente no modelo estudado, ilustando deste modo, a sua reusabilidade em outros sistemas.

No capítulo 5 são descritos os critérios utilizados para se converter a especificação, tanto dos padrões de projeto descritos no capítulo 3, bem como o estudo de caso descrito no capítulo 4 em uma linguagem de programação, a sua conversão é efetuada de maneira manual, cujas etapas são descritas em diversos trabalhos

(FISCHER, 2000; CAVALCANTI; SAMPAIO, 2002; NEVISON, 2001; P.H.WELCH; J.M.R.MARTIN, 2000). Um dos principais requisitos na escolha da linguagem de programação foi a possibilidade de se converter as linguagens de especificação *Object-Z* e CSP de maneira simples e transparente.

No capítulo 6 as conclusões e o resumo das principais contribuições deste trabalho são apresentados.

2 Especificações formais

Basicamente este capítulo pode ser subdividido em cinco seções, a primeira seção tem por finalidade descrever os principais aspectos da utilização de especificações formais, destacando as vantagens e desvantagens na utilização das mesmas (BOWEN; HINCHEY, 2005; HALL, 1990; BOWEN, 1994; BOWEN; HINCHEY, 1995a, 1995b).

A seção 2.2 tem por objetivo descrever as principais características da linguagem de especificação *Object-Z* (SMITH, 1995, 2000a). *Object-Z* é uma extensão de Z (SPIVEY, 1992), desenvolvido para permitir uma especificação formal orientada a objetos. Ela inclui uma construção de classe para encapsular o esquema de estados e todos os esquemas de suas funções que podem afetar as suas variáveis. Um esquema é uma linguagem definida em Z, utilizada para estruturar e descrever informações, encapsulando e nomeando suas informações para reuso (SPIVEY, 1992). Uma classe pode ser utilizada para definir um ou mais componentes de um sistema ou para especificar as interações entre componentes, ou seja, as suas instâncias ou os objetos de suas classes.

A seção 2.3 tem por objetivo descrever as principais características da linguagem de especificação CSP (HOARE, 1985). CSP é uma linguagem de especificação formal que possibilita uma visualização dos aspectos dinâmicos e concorrentes de um sistema computacional, através de operadores e notações descrevendo processos concorrentes e pelas respostas que cada um dos processos executa em relação a eventos comunicados entre eles através de canais ou outras primitivas de sincronização.

A seção 2.4 aborda as considerações utilizadas para se integrar a notação CSP com a *Object-Z*, de modo a integrar em um mesmo formalismo tanto os aspectos dinâmicos como os aspectos estáticos de um sistema computacional. Neste presente trabalho, utiliza-se a integração abordada por Duke e Smith (SMITH, 1995, 2000a, 1995), uma vez que os conceitos de orientação a objetos podiam ser aplicados, sem a necessidade de se introduzir o conceito de tempo na relação existente entre os objetos. Foram utilizados também, os conceitos de sensores e atuadores, como canais CSP, definidas em TCOZ (MAHONY; DONG, 2000, 2002).

A última seção descreve um exemplo de um sistema produtor-consumidor utilizando-

se a notação CSPOZ e a transcrição do exemplo em uma linguagem de programação.

2.1 Especificações formais

A utilização de métodos formais permite o desenvolvimento de sistemas, quando corretamente empregada, em sistemas de alta confiabilidade e segurança (BOWEN, 1993). Através da utilização de modelos matemáticos ela permite ao desenvolvedor verificar importantes propriedades, resolver ambigüidades e detectar erros de análise na fase de concepção do sistema. Sem a utilização de métodos formais, um sistema teria a necessidade de exaustivos testes antes da sua implantação, além desta alternativa ser extremamente custosa, a utilização de testes pode demonstrar a existência de erros, mas não a ausência das mesmas (DAHL; DIJKSTRA; HOARE, 1972). Uma eventual falha poderia causar uma re-implementação de parte ou de todo o sistema, aumentando-se os custos de desenvolvimento, além das questões de segurança não serem garantidas.

A utilização de uma linguagem de especificação torna-se necessária para descrever, analisar e auxiliar na manutenção e no desenvolvimento de sistemas computacionais. A utilização de linguagens de programação como linguagens de especificação tornam-se inadequadas devido à falta de abstrações necessárias para uma especificação, além da descrição de um problema ser efetuada com um nível excessivo de detalhes e na dificuldade de se analisar o código fonte propriamente dito de modo a tirar conclusões a respeito do seu comportamento. Por outro lado, a utilização de uma linguagem natural para se especificar sistemas computacionais torna-se inadequada, devido à falta de precisão e a possibilidade de se introduzir ambigüidades e especificações incompletas, levando conseqüentemente a uma análise inadequada do sistema.

Uma especificação formal de um sistema computacional é baseada em uma linguagem de especificação formal, que por sua vez é baseada em teorias matemáticas, usualmente expressões algébricas, teoria dos conjuntos e lógica, de maneira a possuir uma sintaxe e semântica clara, possibilitando desta forma descrever, modelar e analisar um sistema de forma correta e concisa (HALL, 1990). Uma vez que a definição de sua semântica é baseada em um sistema completo, a análise de suas propriedades pode ser validada de maneira sistemática, possibilitando assim, tirarmos conclusões corretas a respeito do comportamento do sistema.

Uma linguagem de especificação formal não necessita ser executável. Uma especificação formal de uma abstração do sistema deve ser independente em relação a qualquer implementação concreta, ou seja, a especificação descreve o que o sistema faz ao contrário de especificar como ela a implementa. Tais especificações são

necessariamente mais simples e sem a necessidade de descrever nenhum detalhe a respeito dos algoritmos e da estrutura de dados, facilitando a sua verificação. Dessa forma, muitas das definições de especificações formais baseiam-se na idéia de que a demonstração de um sistema abstrato implica, através de sucessivos refinamentos, que o sistema concreto apresente as mesmas características do sistema abstrato.

As especificações formais, além de garantir uma melhor integridade em relação ao sistema podem, por exemplo, apresentar as seguintes características (BOWEN, 1994):

- Uma especificação formal pode fornecer métodos de comunicação entre os desenvolvedores, gerentes e clientes, além de outras pessoas envolvidas no sistema. Ela age como um contrato entre o desenvolvedor e o cliente do sistema, planos de desenvolvimento e formar a base para a documentação,
- Para sistemas que requerem constantes modificações, uma especificação formal pode ser utilizada como um manual de referência para a manutenção do mesmo. Novas propostas de extensão podem ser formalmente especificadas e as interações com o sistema existente determinadas.

Muitas considerações a respeito da importância e da validade de aplicações de métodos formais foram levantadas em Hall (1990), Bowen e Hinchey (1995a), além da utilização correta de especificação formal para o desenvolvimento de sistemas, discutidas e analisadas em Bowen e Hinchey (1995b, 2005). Resumidamente, as seguintes características são desejáveis para uma linguagem de especificação formal:

- Apesar da abstração de detalhes da implementação ser uma qualidade desejável à especificação de sistemas, a possibilidade de se especificar os detalhes formalmente também é bastante interessante, permitindo a utilização de sucessivos refinamentos a partir do nível mais abstrato do sistema até a sua implementação, ou seja, uma linguagem de especificação formal deve possibilitar a descrição de um sistema em diferentes níveis de abstração,
- Devido à presença de não determinismo em alguns sistemas, ou seja, da possibilidade do sistema exibir mais de uma resposta para uma dada entrada, a implementação e definição de um suporte para a presença de não determinismo é essencial para sistemas que apresentam tais características de maneira a minimiza-las. A escolha de um método formal que apresente estas características e a escolha correta do mesmo é essencial para uma boa análise e especificação do sistema,
- Uma propriedade algumas vezes considerada importante para linguagens de especificação é a habilidade de descrever concorrência. Concorrência é essencialmente uma questão inerente à implementação, ou seja, quando mais de

um processo é executado simultaneamente em um sistema. Para permitir o refinamento para se implementar tal característica, é necessário que a linguagem de especificação tenha a habilidade de descrever a ocorrência simultânea de mais de um evento,

- Linguagens de especificação formais também devem ser simples de modo que não comprometam a sua utilização, ou seja, apesar da sua formalização matemática, a linguagem deve ser legível e de rápida compreensão. Uma maneira de se atingir tais características é estruturando a especificação em partes independentes que possam ser lidas e compreendidas isoladamente. Métodos genéricos que possam ser reutilizados também podem aumentar a legibilidade de uma especificação. A sua legibilidade também pode ser alcançada através da inclusão de especificação informal e explanatória através de textos e diagramas, utilizada apenas para auxiliar a compreensão da linguagem formal, não a substituindo.

2.1.1 Combinações de especificações formais

O principal propósito da utilização de especificações formais é implementar uma descrição concisa e correta de softwares. Para sistemas grandes e complexos, a sua descrição formal pode ser alcançada através da utilização de mais de uma linguagem formal de especificação. Apesar da maioria das linguagens de especificações formais utilizadas especificarem sistemas inteiros, alguns, talvez nenhum, consegue modelar todos os aspectos de tais sistemas.

Analogamente há especificações informais utilizados na indústria atualmente, com muitas elaborações e propostas para se unificar as especificações de um software, englobando desse modo os diferentes aspectos que compõem o mesmo. Dentre as inúmeras soluções propostas e utilizadas a mais comumente utilizada é a UML (*Unified Modelling Language*) (RUMBAUGH; JACOBSON; BOOCH, 1998), através da descrição de um sistema utilizando-se diagramas com suas características estáticas e dinâmicas. Entretanto, as estruturações das especificações de sistemas distribuídos modernos separam as especificações em diferentes pontos de vista, dificultando a análise do sistema como um todo. Nos sistemas especificados em UML (RUMBAUGH; JACOBSON; BOOCH, 1998), por exemplo, são utilizados diagramas de classes, seqüência, estados e componentes que, por sua vez, utilizam diferentes linguagens para cada ponto de vista do sistema.

Em relação às especificações formais, um bom exemplo de utilização de combinações de linguagens formais pode ser visualizado na especificação de sistemas distribuídos e concorrentes. Tais sistemas contêm uma quantidade de processos distintos operando concorrentemente e sincronizando através de certos eventos.

Álgebras de processos como CCS (MILNER, 1989), CSP (HOARE, 1985; ROSCOE, 1998; SCHNEIDER, 2000) e π-Calculus (MILNER, 1991, 1999) são adequadas para modelar seus aspectos dinâmicos, as suas interações entre os processos e sua ordem temporal. Linguagens baseadas em estados como Z (SPIVEY, 1992) ou VDM, no entanto, oferecem melhores facilidades para descrever estruturas de dados complexas que podem ser necessárias para descrever os processos.

Para se produzir uma linguagem de especificação integrada, é necessário utilizar uma combinação de linguagens existentes, ou seja, as construções definidas em uma linguagem precisam ser aplicáveis em outras linguagens, possibilitando que as partes sejam relacionadas entre si. Este relacionamento é possível se a construção de uma linguagem possuir uma semântica idêntica à linguagem que se quer integrar, o que muitas vezes não é aplicável devido à falta de definições comuns existentes entre diversas linguagens (SMITH; DERRICK, 2001).

Um dos problemas em se adotar uma única especificação para todos os aspectos de um sistema, particularmente em sistemas concorrentes, é a inexistência de construções na maioria das linguagens baseadas em estados para processos aplicáveis a álgebra de processos, o que ocorre na especificação de linguagens orientadas a objetos como *Object-Z*, uma extensão orientada a objetos de *Z*. Uma das grandes vantagens da orientação a objetos é a de basear-se em uma visão do sistema como um conjunto de objetos distintos interagindo entre si e encapsuladas através de classes, existindo um forte relacionamento entre classes em sistemas orientados a objetos e processos em sistemas concorrentes, dessa forma, as interações entre as instâncias de cada classe definem o comportamento do sistema. Este relacionamento tem sido reconhecido por muitos pesquisadores, tanto na teoria como na prática de orientação a objetos (SMITH, 1995).

Álgebra de processos é uma área da matemática que investiga as relações existentes entre processos concorrentes, basicamente definidas através de uma linguagem algébrica para a especificação de processos e a formulação de suas definições e cálculos para se verificar o comportamento e as relações entre os processos. As principais propostas a respeito de álgebra de processos são CCS (MILNER, 1989) e CSP (HOARE, 1985; ROSCOE, 1998; SCHNEIDER, 2000).

A particular combinação de *Object-Z* e CSP é investigada por diversos pesquisadores como Smith (DERRICK; SMITH, 2003; SMITH; DERRICK, 2001), Fischer (FISCHER, 2000, 1997) e Mahony e Dong (MAHONY; DONG, 2000, 2002). Em geral, a abordagem utilizada para se integrar essas duas linguagens formais consiste de 3 fases. Primeiramente, especifica-se os componentes como classes *Object-Z*. Para manter-se a modularização dos componentes e permitir uma máxima flexibilização para descrever os componentes de processos, cada classe é descrita independentemente em relação aos outros objetos e ao ambiente a que pertence. Os componentes es-

pecificados normalmente não se apresentam em formas que possam ser utilizados por composição através de operadores CSP. A segunda fase consiste em definir-se e modificar-se as interfaces das classes (através da herança de *Object-Z* e operação de renomeação de CSP). Finalmente, na terceira fase, a especificação de todo o sistema é descrita através de operadores CSP, descrevendo as interações existentes entre os objetos.

Através da existência de uma semântica comum para as duas linguagens, o sistema pode ser refinado utilizando-se esta mesma semântica comum a ambas as linguagens, porém, uma vez que introduzimos a semântica CSP às classes *Object-Z* pode-se utilizar os refinamentos baseados em CSP (através de *failures* e *divergencies*) como refinamento de suas relações. Entretanto, a verificação de refinamentos é mais conveniente utilizando refinamentos baseados em estados do que calcular as suas falhas e divergências, através de uma abordagem cujos refinamentos sejam compatíveis com os refinamentos utilizados em CSP (SMITH; DERRICK, 2001; DERRICK; SMITH, 2003).

2.2 Object Z

Uma classe em *Object-Z* é representada sintaticamente por uma estrutura nomeada, podendo possuir parâmetros genéricos. Nesta caixa, podem ser definidos tipos locais e constantes, pelo menos um esquema de estados e um esquema de inicialização de estados associada, e zero ou mais esquemas correspondendo às operações. Uma classe também pode incluir a definição das classes herdadas.

Nas seções 2.2.1-2.2.7 são introduzidas respectivamente os conceitos de visibilidades de variáveis e objetos, agregação de objetos, composição de objetos, herança, definições de variáveis secundárias, polimorfismo e *containment* em *Object-Z*.

A classe em *Object-Z* é um template para objetos, cada objeto de uma classe possui um estado conforme as definições do esquema de estados da classe, e as transições de estado são efetuadas através das operações definidas nos esquemas de operações da classe. Uma classe também é utilizada como tipo, instâncias do tipo são identificadas como referências de objetos desta classe, ou seja, isso possibilita um objeto fazer referência a outros objetos. Uma especificação de um sistema em *Object-Z* consiste de uma definição de classes, seja por herança, um mecanismo de adaptação de classes, seja por modificação ou extensão.

A estrutura básica de uma classe é:

NomedaClasse

definição de tipos

definição de constantes

esquema de estados

esquema de inicialização de estados

esquema de operações

Como exemplo, considere o seguinte exemplo abaixo:

$Stack[T]$

$max : \mathbb{N}$

$max \leq 100$

$itens : \text{seq}\ T$

$\#itens \leq max$

INIT

$itens = \langle\ \rangle$

Push

$\Delta(itens)$

$item? : T$

$\#itens < max$

$itens' = \langle item? \rangle ^\frown itens$

Pop

$\Delta(itens)$

$item! : T$

$\#itens \neq \langle\ \rangle$

$itens = \langle item! \rangle ^\frown itens'$

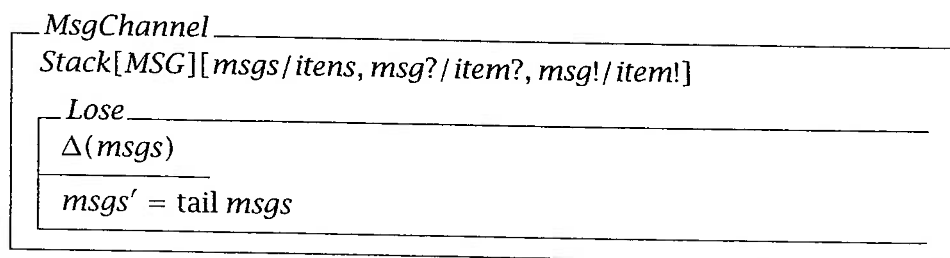
Esta classe possui uma simples constante *max*, definindo o tamanho máximo da pilha e uma única variável de estado *itens* definindo os itens contidos na pilha. Constantes são associadas a um valor fixo que não podem ser alteradas por nenhuma operação interna da classe, porém, os valores das constantes podem ser diferentes para diferentes objetos da classe.

Inicialmente, a pilha é vazia e as operações *Push* e *Pop* habilitam itens de entrar ou sair da pilha, respectivamente. Cada esquema de operação possui um operador Δ definindo a lista de estados que podem ser alterados por esse operador, uma declaração de variáveis de entrada (variáveis que terminam com "?") e declaração de variáveis de saída (variáveis que terminam com "!") e uma parte referente ao

predicado relacionando os pré e pós valores das variáveis de estado (através dos nomes denominados por ').

Uma classe pode ainda herdar as definições de uma ou mais classes. Herança é um poderoso mecanismo para se incrementar especificações permitindo que as necessidades periféricas sejam adiadas (ou seja, implementadas por classes que a herdam) e que as especificações sejam voltadas apenas para o comportamento intrínseco da classe. Os tipos locais e constantes são herdados da classe pai e implicitamente disponibilizadas. Seus esquemas também são disponibilizados ou são implicitamente unidos a esquemas com nomes comuns à classe pai. Como exemplo, considere a seguinte definição de uma mensagem perdida em um canal.

Seja MSG o conjunto de todas as possíveis mensagens.



A classe *MsgChannel* herda de *Stack* com o tipo genérico *T* inicializada com *MSG* e com as variáveis de *itens* renomeado para *msgs* e com os parâmetros de operação *item?* e *item!* renomados para *msg?* e *msg!* respectivamente. No geral, constantes e variáveis de estado, esquemas de operações e parâmetros de operações podem ser renomeados. Objetos de *MsgChannel* comportam-se identicamente a objetos da classe *Stack*, exceto pela existência do operador *Lose* correspondente à perda de mensagem.

2.2.1 Visibilidade dos objetos

Objetos possuem uma visibilidade externa que os identificam ao ambiente que pertencem, e uma visibilidade interna, caracterizada por constantes, estados, inicialização e operações, não visíveis ao meio externo, de modo a encapsular e abstrair as suas complexidades internas.

Sob o ponto de vista externo, se C é uma classe, a declaração $c : C$ declara c cuja variável é uma referência (ou seja, uma identidade) de um objeto da classe C. Referências distintas identificam objetos distintos. Deste modo, C define um conjunto de referências de um potencial conjunto de objetos da classe C. Não existe nenhuma referência que a declaração de um objeto introduz uma referência distinta e, portanto, a um objeto distinto, ou que a declaração de um objeto indique que o mesmo seja inicializado. Por exemplo, $c, d : C$ não implica que c e d se

referem a objetos distintos.

Para assegurar-se a distinção dos objetos, a declaração $c \neq d$ deverá ser incluída no predicado. Se c e d são distintos inicialmente, mas subseqüentemente referem-se a um mesmo objeto, então $c \neq d$ deve ser definida no esquema *INIT* e alguma operação deve, por exemplo, incluir $c' = d$.

Do ponto de vista interno, a semântica de *Object-Z* compreende um conjunto de atributos consistentes com seus tipos definidos e invariantes de classe. Invariantes de classe são condições relacionadas ao estado de uma classe e que devem ser satisfeitos por todo método ou atributo da classe. Uma atribuição de valores pode-se dar através de um conjunto de valores iniciais, através de suas operações, utilizando-se um conjunto de relações com parâmetros de entrada e saída e possivelmente variáveis auxiliares.

O termo $c.att$ define o valor do atributo att do objeto referenciado por c, e $c.INIT$ é um predicado que define se o objeto c está de acordo com as conformidades do esquema de inicialização da classe C. O termo $c.Op$ define a operação na qual o objeto referenciado por c se transforma, de acordo com a definição das operações definidas em C, as operações implementam o único mecanismo de transformação das variáveis internas dos objetos.

2.2.2 Agregação

Uma das vantagens de objetos introduzirem referências a outros objetos é que, por exemplo, um objeto referenciado por c pode envolvê-lo (internamente) sem alterar o valor de c. Isto é bastante significativo quando se modela agregação. Assim, a declaração $sc : \mathbb{P}\, C$ ($\mathbb{P}$ denomina o conjunto potência) modela uma agregação de objetos da classe C que pode envolvê-los internamente sem alterar os valores de cada umd dos objetos de sc, uma vez que somente as referências dos objetos são alteradas. A evolução desse objeto pode ser especificada pelo exemplo abaixo:

Select
$c? : sc$

e utilizando o esquema acima em:

$OP \mathrel{\widehat=} Select \bullet c?.Op$

Esta construção promove a operação Op do objeto nomeado por $c?$ para ser uma operação da classe com o atributo sc. A notação $operation_1 \bullet operation_2$ define o ambiente na medida em que $operation_1$ redefine os valores a serem utilizados por $operation_2$.

Diversos objetos de uma agregação podem cooperar utilizando múltiplas seleções de ambiente como:

$$\begin{array}{|l}
\textit{SelectTwo} \\
\hline
c_1?, c_2? : sc \\
\hline
c_1? \neq c_2? \\
\hline
\end{array}$$

e utilizando como:

$Together \mathrel{\widehat=} SelectTwo \bullet (c_1?.Op \land c_2?.Op)$

Uma referência a um objeto é apenas listada com Δ se a referência altera os valores do objeto e refere-se a um outro objeto. Uma referência a objetos como o exemplo acima, sc,´possui somente o parâmetro Δ se o conjunto é alterado, ou seja, se novas referências são adicionadas, retiradas ou substituídas.

2.2.3 Composição

Considere a classe *StackPair* que possui duas referências para Stack de números naturais:

$$NatStack == Stack[\mathbb{N}][nats/itens, nat?/item?, nat!/item!]$$

$$\begin{array}{|l}
\textit{StackPair} \\
\hline
\quad s_1, s_2 : NatStack \\
\quad \hline
\quad s_1 \neq s_2 \\
\quad \#s_1.nats \leq \#s_2.nats \\
\\
\quad \textsc{Init} \\
\quad s_1.INIT \land s_2.INIT \\
\\
Push_1 \mathrel{\widehat=} s_1.Push \\
Push_2 \mathrel{\widehat=} s_2.Push \\
Pop_1 \mathrel{\widehat=} s_1.Pop \\
Pop_2 \mathrel{\widehat=} s_2.Pop \\
PushBoth \mathrel{\widehat=} Push_1 \land Push_2 \\
Transfer \mathrel{\widehat=} (Pop_1 \parallel Push_2) \setminus (nat!) \\
PushOne \mathrel{\widehat=} Push_1 \mathbin{\Box} Push_2 \\
TransferAll \mathrel{\widehat=} Transfer \mathbin{⨟} TransferAll \mathbin{\Box} [s_1 = \langle\,\rangle] \\
\hline
\end{array}$$

Pode-se perceber que s_1 e s_2 são distintos, uma vez que as suas referências são distintas. É definido que o tamanho da primeira pilha não deve exceder o tamanho da segunda pilha. As duas pilhas são inicializadas através do esquema de inicialização da classe.

A operação $Push_1$ promove a operação *Push* de s_1 para que seja uma operação de *StackPair*. Similarmente, $Push_2$, Pop_1, Pop_2 são promovidas.

Como s_1 e s_2 não aparecem juntos ao operador Δ, elas continuam a se referir a seus respectivos objetos.

A operação *PushBoth* ilustra uma operação com a conjunção '$\wedge$'. Neste exemplo, cada uma das operações de *Push* funciona independentemente e com a mesma funcionalidade, exceto pelo fato de que um mesmo valor *nat*? é aplicado a ambas as pilhas. A operação de conjunção condiciona a operação de modo a utilizar as mesmas variáveis de entrada e condições em ambas as operações, ou seja, no exemplo acima a variável *nat*? é utilizada para ambas as funções $Push_1$ e $Push_2$.

A operação *Transfer* ilustra o operador de paralelismo '$\|$'. Sincronizando as entradas e saídas comuns das operações. Conceitualmente o item retirado através do operador Pop_1 é inserido na pilha s_2 através do operador $Push_2$. Como o operador de conjunção, ela é comutativa e, portanto, suporta comunicações em ambas as direções. Entradas que recebem a comunicação (no caso *nat*?) são escondidas ao meio externo, mas para se tornar o operador de paralelismo associativo, as saídas não são escondidas. Saídas não relevantes ao ambiente como *Transfer nat*!, devem ser explicitamente escondidas. Como o operador de conjunção '$\wedge$', saídas de mesmo tipo são utilizadas em ambas as operações. Saídas residuais (aquelas que não possuem par com as variáveis de entrada e, portanto, não escondidas) são utilizadas em ambas as operações.

Uma definição alternativa de *Transfer* seria:

$Transfer \mathrel{\widehat{=}} s_1.Pop \mathbin{⨟} Push_2$

Esta definição ilustra o operador seqüencial '$⨟$'. Ele forma uma composição dos operadores de *Transfer*, sendo que o operador é não comutativo, e não associativo devido ao mecanismo de comunicação existente neste operador. A comunicação é efetuada da esquerda para a direita, sendo que as suas variáveis são escondidas do ambiente, ou seja, as saídas do operador da esquerda são sincronizadas com as entradas do operador da direita que pertençam ao mesmo tipo, além do fato de serem escondidas ao ambiente. Portanto, na composição seqüencial dos operadores no exemplo acima (*Transfer*), *nat*? proveniente da operação $Push_2$ é sincronizada com a variável *nat*! do operador $s_1.Pop$. Entradas residuais com o mesmo tipo são sincronizadas com as saídas residuais. A semântica de sequenciação requer uma notação de estados intermediários.

A operação *PushOne* ilustra uma operação de escolha '$\Box$'. O operador indica uma escolha não determinística de uma das operações que satisfaçam as pré-condições, deste modo, a operação é determinística se e somente se uma pré-condição é satisfeita e a outra falha esteja relacionada à não adequação desta pré-condição. Dependendo dos estados de s_1 e s_2, existem três possíveis saídas de *PushOne*, o operador falha (ambas as pré-condições não são satisfeitas, ou seja, as duas pilhas estão

cheias e s_1 é do mesmo tamanho que s_2), *nat*? é inserida em s_1 ou *nat*? é inserida em s_2. Existe uma certa noção de comunicação entre os operadores, uma vez que uma das opções é selecionada. O operador é comutativo e associativo.

A operação *TransferAll* ilustra uma composição seqüencial de maneira recursiva. A construção é determinística uma vez que ou *Transfer* ou $Empty_1$ é verdadeira. O efeito de *TransferAll* é colocar todos os itens de s_1 em uma seqüência reversa em s_2.

Todas as operações em *Object-Z* são relações ou composições de relações, ou seja, as operações são atômicas, incluindo-se operações recursivas como a relação *TransferAll*.

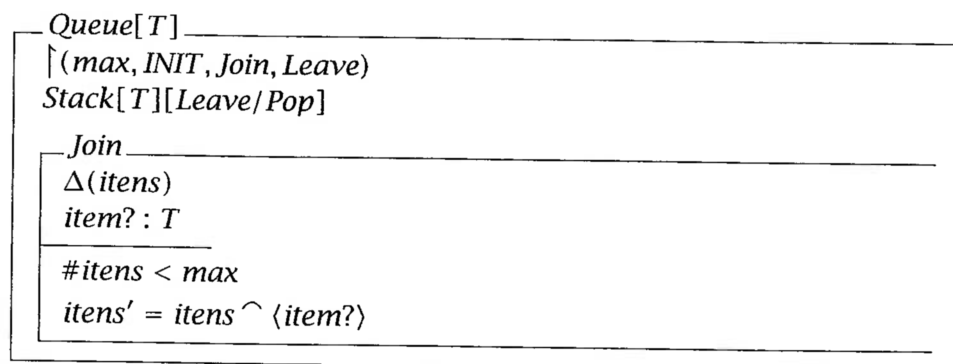
As precedências dos operadores de conjunção e paralelismo são iguais. A composição seqüencial possui uma precedência menor, e a escolha possui uma precedência menor do que seqüência.

2.2.4 Herança

Herança é um mecanismo de especificação incremental, em que uma nova classe pode derivar-se de uma ou mais classes.

Essencialmente, todas as definições baseiam-se no fato de que as definições de tipos e constantes da classe herdada são adicionadas na classe derivada. Os estados e os esquemas de inicialização da classe herdada são redefinidos ou não. Na classe derivada as operações com o mesmo nome são redefinidas. Conflitos de nomes, que poderiam levar a um mau entendimento da classe, podem ser resolvidos por renomeação quando herdadas.

Considere a derivação da classe genérica *Stack[T]*.



O símbolo de projeção '↾' indica visibilidade, isto é, introduz a lista de itens acessíveis externamente, quando essa mesma lista é omitida, todos os itens pertencentes à classe são visíveis.

A operação *Pop* é renomeada para um nome mais apropriado como *Leave*, e

a operação *Push* renomeada para *Join*, cuja operação insere o elemento *item*. A operação *Push* é herdada, porém, devido à sua operação não ser característica de uma fila, a sua visibilidade foi suprimida. Devido ao fato da classe *Queue[T]* derivar-se da classe *Stack*, a operação *Push* poderia ser herdada também.

A derivação de *Queue[T]* de *Stack[T]* ilustrou situações em que nem sempre a classe derivada preserva as operações da classe pai, uma vez que a pilha possui a natureza de primeiro a entrar - último a sair e a fila possui a natureza de primeiro a entrar - primeiro a sair.

Heranças múltiplas são semanticamente equivalentes a uma corrente de derivações simples.

Redefinições de operações são afetadas pela definição de operações de mesmo nome da classe derivada.

2.2.5 Variáveis secundárias

As variáveis descritas até agora são primárias, ou seja, só poderiam ser alteradas através de operadores cuja lista Δ as incluíssem explicitamente. É conveniente, porém, introduzir variáveis de estados adicionais para aumentar a legibilidade da especificação. Por exemplo, considere a adição da variável *size* para o esquema de estados da classe *Stack*:

$$\begin{array}{|l}
itens : \mathrm{seq}\, Item \\
size : \mathbb{N} \\
\hline
size = \#itens \\
size \leq max
\end{array}$$

Os possíveis valores de *size* incluem: $size = 0$, $size > 0$, $size \leq max$. A variável de estado *size* adiciona nenhuma informação a mais do que a já existente no sistema em relação a *itens*, porém, devido a alteração do valor de *size* através das operações de *Push* e *Pop*, é necessário aparecer na lista Δ. Para preservar a dependência das variáveis sem a necessidade de inclui-las na lista Δ que contém todas as variáveis dependentes, a noção de variáveis secundárias é introduzida.

Variáveis secundárias basicamente dependem das variáveis primárias e são implicitamente incluídas na lista Δ. Sintaticamente, variáveis secundárias são introduzidas através da declaração Δ que sugere a sua inclusão implícita em todas as listas Δ.

Retornando ao exemplo *Stack*e introduzindo *size* como uma variável secundária, o esquema anterior poderia ser reescrito como:

$itens : \text{seq}\ Item$
Δ
$size : \mathbb{N}$

$size' = \#itens'$
$size \leq max$

A variável secundária *size* é implícita à lista Δ das operações *Push* e *Pop*.

A declaração $size' = \#itens'$ aplica-se a toda operação, preservando-se deste modo a dependência de *size* e *itens* para todas as operações.

2.2.6 Polimorfismo

Polimorfismo é um mecanismo que permite uma variável ser declarada e cujo valor pode pertencer a qualquer objeto pertencente a uma coleção pré-definida de classes. Em relação a linguagens orientadas a objetos, é comum esta coleção ser formada por todas as classes derivadas de uma classe comum ancestral.

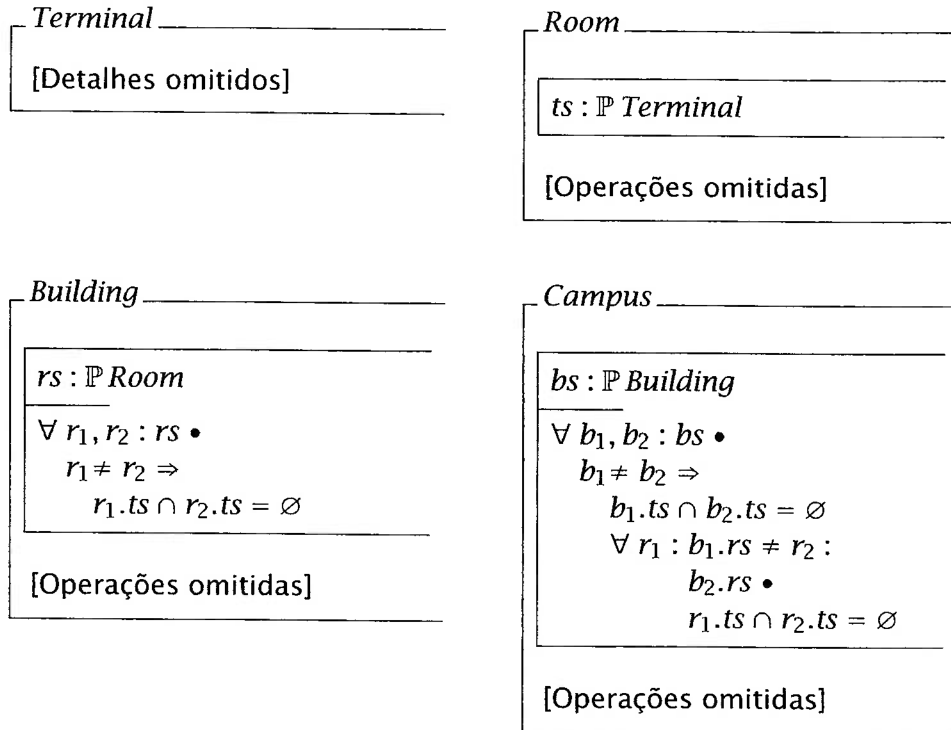
A declaração $c :\downarrow C$ define c como uma referência de C ou outra classe derivada de C. A definição interna de c depende da instância da classe C na qual foi definida, ou seja, utilizando-se o histórico de modelos de classes, $\downarrow C$ é a união de todas as classes derivadas de C, incluindo C.

Polimorfismo em *Object-Z* permite que uma variável seja declarada de forma a associar-se a mais de um tipo. Portanto, algumas regras devem ser obedecidas quando se utilizam variáveis polimórficas. Especificamente, estas regras requerem que qualquer expressão envolvendo variáveis polimórficas possa ser aplicada a qualquer classe possível de ser herdada. A responsabilidade desta especificação pode ser reduzida se a herança for restringida de maneira que a assinatura seja compatível com a classe herdada.

Uma classe com a assinatura compatível com uma classe herdada pode ser utilizada em qualquer contexto em que essa classe herdada puder ser utilizada, ou seja, qualquer chamada de operação efetuada para a classe herdada também pode ser efetuada para a classe cuja assinatura seja compatível. Portanto, se uma classe derivada mantiver a compatibilidade de assinatura, a especificação apenas precisa garantir que a variável polimórfica do objeto do topo da hierarquia da classe possa ser utilizada.

2.2.7 *Containment*

Considere a situação na qual um campus contém um conjunto de prédios, cada prédio contém um conjunto de quartos e cada quarto contém uma quantidade de terminais. A especificação em *Object-Z* seria:



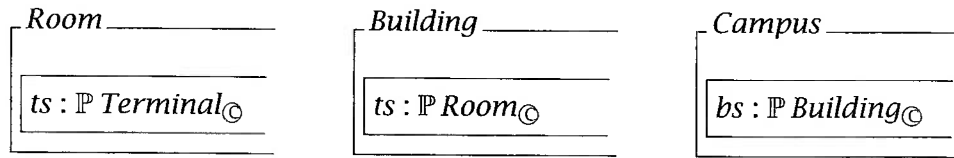
A classe *Building* especifica que nenhum terminal pode estar em dois quartos distintos em um prédio. Similarmente a classe *Campus* especifica que nenhum quarto pode estar em dois prédios distintos no campus. Apesar do fato da classe *Building* especificar que nenhum terminal pode estar em dois quartos distintos, ela somente se aplica a quartos de um dado prédio, ela não se aplica a prédios diferentes, e devido a essa inconsistência a conjunção:

$\forall r_1 : b_1.rs \neq r_2 : b_2.rs \bullet r_1.ts \cap r_2.ts = \varnothing$

foi introduzida na classe *Campus*.

Claramente, capturando-se as propriedades do sistema através das descrições desse conjunto de classes pode-se tornar um tanto trabalhoso, principalmente ao se tratar de sistemas grandes e complexos. O mais simples seria uma invariante global que descrevesse o fato que quartos em qualquer lugar possuíssem terminais distintos e prédios distintos contêm quartos distintos. A condição que quartos distintos possuem terminais distintos não é uma invariante interna pertencente à classe *Room*, mas sim, uma invariante do sistema contendo objetos *Room*. Ou seja, seria interessante que tal informação estivesse dentro das definições da classe *Room*.

Se o papel de um atributo é sempre identificar diretamente objetos contidos, ele pode ser indicado na declaração do atributo através da letra '$_{\copyright}$' do tipo apropriado, removendo-se a necessidade de escrever explicitamente os predicados dos objetos contidos. No exemplo anterior, utilizando-se esses predicados, o sistema se tornaria:



O parâmetro '$_{\copyright}$' é utilizado no tipo de atributo ao invés do atributo, uma vez que o atributo pode identificar uma estrutura de dados complexa em vez de uma referência do objeto. Por exemplo, a declaração:

$ts : \mathbb{P}\, Terminal_{\copyright}$

na classe *Room* declara *ts* como um conjunto, e não uma referência a um objeto. Ou seja, *ts* é um conjunto de referências de objetos *Terminal*, o termo '$_{\copyright}$' implica que essas referências são relacionadas a objetos contidos em outros objetos.

Em geral, duas propriedades de objetos contidas são aplicadas quando se utiliza a abreviação '$_{\copyright}$'

- Nenhum objeto o contém, mesmo direta ou indiretamente.
- Nenhum objeto é contido diretamente em dois objetos distintos.

2.3 CSP

CSP (HOARE, 1985) é uma especificação formal na qual é possível uma visualização dos aspectos dinâmicos e concorrentes de um sistema computacional, através de operadores e notações descrevendo processos concorrentes e através das respostas que cada um dos processos executa em relação a eventos comunicados entre eles, por canais ou outras primitivas de sincronização.

A primeira versão da linguagem de especificação CSP foi definida por C. A. R. Hoare (HOARE, 1978) em 1978 e o primeiro livro de referência, publicada em 1985 (HOARE, 1985), cuja semântica e notações foram alteradas levemente em Roscoe (1998), em que se baseia a notação atual. Fazendo-se ainda uma análise cronológica de CSP, cita-se a introdução, análise e definição da semântica com a introdução do conceito de tempo (SCHNEIDER, 2000) e da introdução do conceito de prioridades em $\mathcal{CSPP}$ (LAWRENCE, 2003b, 2002, 2003a, 2004).

Uma das grandes vantagens na linguagem de especificação CSP é que ela de-

fine notações e especificações muito próximas de uma linguagem de programação como occam (HOARE, 1988), além de possibilitar ao mesmo tempo analisar e especificar um sistema em um nível mais abstrato, possibilitando através de sucessivos refinamentos uma especificação mais concreta do sistema real.

O objetivo do CSP é implementar uma especificação simples, eficiente, funcional, escalável e distribuída envolvendo sistemas. As primitivas são baseadas na álgebra de Hoare, *Communicating Sequential Processes*, permitindo as aplicações serem modeladas de maneira direta e formal. Ela permite a definição de concorrência em todos os níveis de um sistema de maneira bastante abstrata. Por exemplo, entre equipamentos funcionando paralelamente (seja ele um "cluster"ou um sistema distribuído na Internet), programas *multithreading* ou sincronização entre protocolos de comunicações. Os mesmos conceitos e teorias são utilizados desconsiderando-se os efeitos físicos da concorrência, seja ela, por exemplo, uma memória compartilhada ou sincronização de eventos em um processo produtivo.

As principais idéias do CSP, como definida por Hoare são:

1. Os processos são acoplados apenas a eventos.
2. Os processos paralelos ou concorrentes podem trabalhar juntos através de sincronização das suas respectivas entradas e saídas de seus processos. A comunicação, como proposto por Hoare, somente pode ocorrer se o processo A indicar que está pronto para comunicar-se com o processo B e ao mesmo tempo em que o processo B indicar que está pronto para receber dados do processo A. Caso uma dessas condições seja falsa, o processo associado irá esperar até que a comunicação seja habilitada (*"rendez-vous"*).
3. Os processos paralelos ou concorrentes se comunicam de maneira instantânea; não existe nenhuma consideração a respeito do intervalo durante os mesmos, ou seja, eles ocorrem atomicamente. Esta propriedade implica em *"safety"* durante condições de corrida.

A subseção 2.3.1 faz uma introdução à sintaxe de CSP, descrevendo as principais características de um sistema concorrente e a introdução do conceito de eventos e processos em CSP. A subseção 2.3.2 descreve os operadores responsáveis pelo controle de fluxo de processos em CSP. A subseção 2.3.3 introduz o conceito de *Traces*, aspecto importante na análise do sistema abordado nas subseções 2.3.4, 2.3.5, 2.3.6.

2.3.1 Sintaxe CSP

CSP possui uma grande quantidade de operadores utilizados para descrever características dinâmicas de um sistema. Ela possui primitivas básicas descrevendo aspectos de seqüenciação, concorrência e não determinismo de um sistema, além de operadores mais complexos descrevendo aspectos como renomeação, tempo, prioridade, etc. Um maior detalhamento da sua sintaxe pode ser encontrado em Hoare (1985), Roscoe (1998) e Schneider (2000).

2.3.1.1 Eventos e Processos

O elemento básico de comunicação no CSP é o Evento. Eventos podem ser de natureza atômica ou associados a dados através de canais de comunicação de entrada e saída. Por exemplo, $a \rightarrow b$ são eventos atômicos, enquanto que $out!5$ e $in?x$ representam eventos cuja saída é o valor 5 e entrada o valor x, respectivamente.

Um conjunto de nome de eventos pertinentes à descrição de um objeto, processo ou sistema é chamado de alfabeto, normalmente definido como Σ, quando relacionado a todos os nomes de eventos do sistema ou, por exemplo, αP para o alfabeto dos nomes de eventos relacionados a um dado processo P.

Um processo é uma entidade que possui um comportamento, com uma interface particular na qual ela interage com o ambiente. Uma vez que um processo interage com outros processos através desta interface, o comportamento do processo em si é determinado pelo comportamento desta interface, ou seja, os comportamentos internos referentes ao processo são abstraídos e a análise é feita particularmente sobre a sua interface. A interface, por sua vez, é descrita pelo seu alfabeto.

Por exemplo, uma impressora pode aceitar um trabalho, imprimir o documento e por fim, parar:

$PRINTER = accept \rightarrow print \rightarrow \textsc{Stop}$

e o seu alfabeto, definido como:

$\alpha PRINTER = \{accept, print\}$.

Em CSP os seguintes processos primitivos são definidos abaixo:

- STOP - processo em que nada ocorre (máquina quebrada).
- SKIP - processo terminado com sucesso.
- DIV - processo com *livelock*, ou seja, um processo em que somente ocorrem eventos "invisíveis"ao ambiente externo.

2.3.1.2 Prefixo

Para se expressar que um dado processo representa a ocorrência de um evento e subseqüentemente, funcionando como um novo processo, é utilizado o operador de prefixo. Seja P um processo e $a \in \Sigma$ é um evento qualquer, a denotação do operador de prefixo é dada da seguinte maneira:
$a \rightarrow P$

Um evento pode ser um evento de comunicação, que é representado pelo par $c.v$, em que c é o nome do canal, e v o valor da mensagem transmitida.

Input ou eventos de leitura são escritos como $c?x \rightarrow P(x)$ e *Output* ou eventos de saída são escritos como output: $c!x \rightarrow P$. A ocorrência de um evento de leitura está intrinsecamente relacionada com a ocorrência de um evento de escrita, ou seja, as ocorrências dos mesmos ocorrem somente quando os dois processos, no caso os processos de leitura e escrita, estão aptos a executar a comunicação.

2.3.1.3 Recursividade

Algumas vezes, é necessária a utilização de um comportamento repetitivo para a especificação de um processo. A recursão em CSP pode ser atingida através da referência ao nome do processo no lado direito da equação que a define. Na equação $P = a \rightarrow P$, P executa o evento a e retorna a se comportar como P, ou seja, através de substituição, o mesmo processo P pode ser redefinido como $P = a \rightarrow a \rightarrow P$.

Este método de definição de recursividade de um processo somente funciona se ao menos um dos eventos é prefixado para todas as ocorrências recursivas de um processo. A descrição de um processo que começa com um prefixo é dito como um processo guardado por um evento. Se $F(X)$ é uma expressão guardada contendo o processo X e A é o alfabeto de X, então pode-se dizer que a solução da equação:

$$X = F(X)$$

Possui uma única solução com o alfabeto A, e muitas vezes é conveniente utilizar a notação λ de ponto fixo. Um ponto fixo é uma solução de uma equação recursiva, ou seja, pode-se definir a semântica da recursão usando ponto fixo. Como a função recursiva possui apenas uma única solução e, conseqüentemente a menor solução, é utilizada a notação de *least fixed point* μ.

A equação, portanto, poderia ser reformulada como:

$P = \mu X \bullet a \rightarrow X$ ou $P = \mu X : \{a\} \bullet a \rightarrow X$, sendo que a definição do alfabeto, no caso $\{a\}$ pode ser explícita ou não. Nesta dissertação, foi adotada a versão implícita do seu alfabeto, ou seja, a notação $P = \mu X \bullet a \rightarrow X$.

Exemplos:

$P1 = a \rightarrow b \rightarrow P1$ ou $P1 = \mu X \bullet a \rightarrow b \rightarrow X$

$P2 = up \rightarrow down \rightarrow up \rightarrow down \rightarrow P2$ ou

$P2 = \mu X \bullet up \rightarrow down \rightarrow up \rightarrow down \rightarrow X$

Ou seja, *least fixed point* possui a mesma expressividade de funções recursivas, uma vez que:

$\mu X \bullet F(X) = F(\mu X \bullet F(X))$

ou

$\mu X \bullet P = P(\mu X \bullet P/X)$

Uma outra maneira de se definir recursividade em CSP é através de uma recursividade mútua, ou seja, a definição de um processo através da sua relação com outros processos.

Exemplo:

$LIGHT = on \rightarrow ON$

$ON = off \rightarrow LIGHT$

2.3.1.4 Escolhas

O operador de escolha permite que um processo possa escolher uma entre uma quantidade finita e pré-definida de possíveis eventos, em que o operador oferece um conjunto de eventos possíveis de serem executados em um determinado momento.

Na definição do operador de escolhas, uma escolha pode ser definida como uma escolha externa, na qual o controle sobre a escolha é efetuado externamente ao processo, ou pode ser definida como uma escolha interna, em que o processo de escolha é executado internamente.

A sintaxe para o operador de escolha externa é definida como:

$P1 \square P2$

enquanto que para o operador de escolha interna é definida como:

$P1 \sqcap P2$

Devido à natureza das escolhas, pode-se dizer que a escolha externa possui um comportamento determínistico em relação ao sistema, enquanto que a escolha interna possui uma natureza não determinística.

Ainda em relação às escolhas, uma nova extensão ao CSP foi introduzida por

Lawrence ($\mathcal{CSPP}$) (LAWRENCE, 2003b, 2002, 2003a, 2004), na qual o conceito de prioridades é inserido na definição de escolhas, bem como a base algébrica em relação a processos envolvendo prioridades.

$\mathcal{CSPP}$ introduz o conceito de prioridades, por exemplo:

$(a \rightarrow \text{STOP}) \overleftarrow{\Box} (b \rightarrow \text{STOP})$

O arco no exemplo acima aponta para o processo de maior prioridade. Este processo se comporta como o exemplo sem prioridades $(a \rightarrow \text{STOP}) \Box (b \rightarrow \text{STOP})$, com a diferença que caso os eventos *{a e b}* estejam disponíveis, o processo escolhe o evento a em detrimento do evento b.

$\mathcal{CSPP}$ também inclui uma versão em que existe a simetria na escolha externa, como no exemplo anterior, $(a \rightarrow \text{STOP}) \overleftrightarrow{\Box} (b \rightarrow \text{STOP})$, que se comporta basicamente como um processo na qual um único evento é oferecido ao processo, porém, quando um conjunto de eventos $\{a, b\}$ é oferecido, ela se comporta como o exemplo sem prioridades de CSP, tornando-se desta maneira uma escolha não determinística.

Um outro tipo de não determinismo é gerado considerando-se o exemplo $(a \rightarrow \text{STOP}) \Box (b \rightarrow \text{STOP})$ no $\mathcal{CSPP}$. Não importa se $\Box$ é substituído por $\overleftarrow{\Box}$, ou $\overrightarrow{\Box}$, ou $\overleftrightarrow{\Box}$ caso apenas um único evento seja disponível ao processo, as três notações apresentadas são equivalentes ao CSP clássico. Mas caso um conjunto de eventos $\{a, b\}$ esteja disponível, pode-se concluir que os três operadores CSPP citados acima são refinamentos de $\Box$ da seguinte maneira:

$$
\begin{aligned}
(a \rightarrow \text{STOP}) \Box (b \rightarrow \text{STOP}) = & (a \rightarrow \text{STOP}) \overleftrightarrow{\Box} (b \rightarrow \text{STOP}) \sqcap \\
& (a \rightarrow \text{STOP}) \overleftarrow{\Box} (b \rightarrow \text{STOP}) \sqcap \\
& (a \rightarrow \text{STOP}) \overrightarrow{\Box} (b \rightarrow \text{STOP}).
\end{aligned}
$$

2.3.1.5 Concorrência

Quando dois processos são executados concorrentemente, eles constituem uma combinação paralela. Cada processo executa de maneira independente, de acordo com os padrões previamente escritos, mas as alterações efetuadas no mesmo podem afetar ou não os demais processos.

O modo como processos paralelos CSP interagem está intimamente ligado com os eventos e sincronizações. Qualquer evento que apareça na interface de ambos os processos deve envolver-los caso o evento ocorra. Este mecanismo de como os processos paralelos interagem é definido pela sincronização dos processos de maneira simétrica e instantânea, e ocorre apenas quando ambos os participantes aceitam as condições de paralelismo simultaneamente.

2.3.1.6 Interface paralela

A sintaxe definida para processos paralelos é:

$P1 \ |[A | B]| \ P2$.

Onde o processo P1 é sincronizado com o processo P2 através do alfabeto de eventos A, enquanto que o processo P2 é sincronizado através do alfabeto B.

Por exemplo:

$P1 = a \rightarrow b \rightarrow P1$

$P2 = a \rightarrow c \rightarrow b \rightarrow P2$

$P1 \ |[\{a, b\} | \{a, b, c\}]| \ P2$.

O processo acima pode executar somente o evento a, sendo que logo após a execução do evento a, somente o evento c é possível de se executar, logo após a execução de c a sincronização através do evento b é efetuado, ou seja, o processo *P1* é bloqueado até que o processo *P2* execute o evento c.

Uma combinação paralela de sincronização entre processos híbridos pode ser definida como aqueles nos quais os eventos aparecem numa interface comum A, descrita como:

$P1 |[A]| P2$

Ou seja, os dois processos sincronizam em relação aos eventos definidos por A.

2.3.1.7 *Interleaving*

A execução de processos concorrentes em que nenhum tipo de sincronização é requerido, ou seja, são independentes entre si, é definida como:

$P1 \ ||| \ P2$

O operador de *Interleaving* é basicamente um caso particular de paralelismo no qual o seu alfabeto de sincronização X é um conjunto vazio.

2.3.2 Controle de Fluxo

2.3.2.1 *Hiding*

Quando uma coleção de processos é combinada em um sistema, existirão sempre componentes entre eles que deverão ser internos, e cuja existência na interface do sistema pode ser inapropriado. Um conjunto de eventos A pode ser encapsulado

por um processo P usando-se a seguinte notação:

$$P \setminus A$$

2.3.2.2 Renomeação de Eventos

Um processo padrão de comunicação pode ser descrito em termos de eventos particulares e é possível obter novos processos através da renomeação desses eventos, permitindo-se o reuso de descrições e comportamentos de outros processos sem a necessidade de reescrevê-los.

$$f(P) \xrightarrow{f(a)} f(P')$$

2.3.2.3 Composição Sequencial

Um processo, logo após a sua invocação, retém internamente a ela o controle de execução do seu processamento. Muitas vezes, esse controle de execução pode-se perdurar indefinidamente, bem como passar o controle para um segundo processo, logo após a sua execução ou porque o segundo processo demande o controle de execução. O comportamento do operador de composição seqüencial é definido pelo operador ; .

Por exemplo, seja P um processo bem definido e separável em dois processos $P1$ e $P2$ distintos, sendo que a ocorrência de $P2$ esteja intrinsecamente relacionada com o término de $P1$, a composição seqüencial pode ser utilizada e definida como:

$$P = P1 \; ; \; P2$$

Ou um exemplo mais prático no qual logo após o evento de escolha de um produto seja sucedido por um evento de pagamento do produto em um processo de compra, definido como:

$$COMPRA = ESCOLHA \; ; \; PAGAMENTO$$

2.3.2.4 Interrupção

O controle de um processo $P1$ pode ser passado para outro processo $P2$ através de uma interrupção, ou seja, o controle de $P1$ pode ser removido a qualquer ponto pelo processo $P2$.

$$P1 \triangledown P2$$

2.3.3 *Traces*

Um *trace* é uma seqüência visível de comunicações efetuadas por um processo em um determinado momento de execução, ou seja, uma classe particular de seqüências finitas de eventos efetuados em $\Sigma^{\checkmark}$, onde $\Sigma^{\checkmark}$ corresponde a todas as seqüências finitas de eventos possíveis mais o evento de terminação $\checkmark$, ou seja, $\Sigma \cup \{\checkmark\}$. Assim, pode-se considerar que um *trace* é basicamente uma anotação de todos os eventos observáveis que ocorreram até um determinado momento e anotados, de modo que em um dado momento é impossível observarmos a ocorrência de dois eventos simultaneamente.

Ou seja:

$traces = \{tr \mid \sigma(tr) \subseteq \Sigma^{\checkmark} \land \#tr \in N \land \checkmark \notin \sigma(init(tr))$

Um *trace* é basicamente definido por uma seqüência de eventos separados por vírgula e delimitados por $\langle$ e $\rangle$.

Ex.: $\langle a1, a2, a3, \ldots, an\rangle$ é uma seqüência contendo $a1, a2, a3, \ldots, an$ na mesma ordem apresentada.

Algumas operações envolvendo *traces*:

- Concatenação: Concatena *traces* em *traces* maiores

  Exemplo:

  $s ^\frown t$ é a concatenação de s e t: $\langle a, b\rangle ^\frown \langle b, a\rangle = \langle a, b, b, a\rangle$

- Restrição: A expressão $tr \upharpoonright A$ restringe o trace tr apenas a eventos contidos em A.

  Exemplo:

  $\langle start, exercise, exercise, end\rangle \upharpoonright \langle start, end\rangle = \langle start, end\rangle$

- *Head* e *Tail*: Se tr é um *traces* não vazio, o seu primeiro elemento é definido como tr_0 e o *traces* formado por todos os eventos depois do seu primeiro elemento é definido como tr'.

  Exemplo:

  $\langle coin, choc\rangle_0 = coin$ e $\langle coin, choc\rangle' = \langle choc\rangle$

- Ordenação: Um *trace* tr_0 é um prefixo de um *traces* tr_1 se existe alguma extensão tr_2 de tr_0 na qual $tr_0 ^\frown tr_2 = tr_1$. Define-se como $tr_0 \leq tr_1$.

  Exemplo:

  $\langle a, b, c\rangle \leq \langle a, b, c, d\rangle$

2.3.3.1 Regras na utilização de *Traces*

Existe uma quantidade finita de regras para se calcular os *traces* para cada operador CSP existente.

Exemplos:

- Operadores fundamentais

$traces(\text{STOP}) = \{\langle\rangle\}$

$traces(a \to P) = \{\langle\rangle\} \cup \{\langle a\rangle ^\frown s \mid s \in traces(P)\}$

$traces(P \Box Q) = traces(P) \cup traces(Q)$

$traces(P \sqcap Q) = traces(P) \cup traces(Q)$

- Paralelismo

$traces(P \parallel Q) = traces(P) \cap traces(Q)$

$$traces(P \; |[X \mid Y]| \; Q) = \{s \in (X \cup Y)^* \mid s \upharpoonright X \in traces(P) \wedge s \upharpoonright Y \in traces(Q)\}$$

$traces(P \; ||| \; Q) = \bigcup\{s \; ||| \; t \mid s \in traces(P) \wedge t \in traces(Q)\}$

- *Hiding* e Renomeação

$traces(P \setminus A) = \{s \upharpoonright \Sigma A \mid s \in traces(P)\}$

$traces(f[P]) = \{f(s) \mid s \in traces(P)\}$

- Operador Seqüêncial

$traces(\text{SKIP}) = \{\langle\rangle, \langle\checkmark\rangle\}$

$$traces(P; \, Q) = (traces(P) \cap \Sigma^*) \cup \{s ^\frown t \mid s ^\frown \langle\checkmark\rangle \in traces(P) \wedge t \in traces(Q)\}$$

2.3.3.2 Refinamento de *Traces*

Uma outra maneira de se realizar uma especificação através de *traces* é criando um processo que possua uma quantidade máxima de *traces* satisfazendo a especificação e testá-la através deste mesmo refinamento entre a especificação do processo P e a implementação proposta Q.

Processo Q refina o processo P (definido como $P \sqsubseteq_T Q$) se

$$traces(Q) \subset traces(P)$$

Pode-se perceber que o processo se torna cada vez mais refinado à medida em que ela possui uma menor quantidade de *traces*. Isto ocorre uma vez que quanto

menor a quantidade de comportamentos que um processo possui, menor é a possibilidade dele violar alguma especificação.

$$P \sqsubseteq_T Q \equiv P =_T Q \sqcap P$$

$$Q \; sat \; R(tr) \Leftrightarrow P_R \sqsubseteq_T Q$$

Onde P_R é o processo não determinístico satisfazendo R para qualquer *traces* existente.

2.3.3.3 Semântica de *Traces*

A extração de informações a respeito de *traces* e das regras de transições de processos possibilita a visualização de relação existente entre os processos. O modelo de *traces* considera os processos diretamente em termos de seus eventos, mudando a abordagem de análise dos processos CSP para um nível mais abstrato. Todos os operadores da linguagem podem ser entendidos neste nível de abstração, um exemplo abaixo é dado para a análise de *traces* efetuados por dois processos paralelos:

$P1 = a \rightarrow b \rightarrow \text{STOP}$

$traces(P1) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$

$P2 = b \rightarrow c \rightarrow \text{STOP}$

$traces(P2) = \{\langle\rangle, \langle b\rangle, \langle b, c\rangle\}$

$P1 \; |[\, a, b \mid b, c \,]| \; P2$

$= a \rightarrow b \rightarrow \text{STOP} \; |[\, a, b \mid b, c \,]| \; b \rightarrow c \rightarrow \text{STOP}$

$= a \rightarrow (b \rightarrow \text{STOP} \; |[\, a, b \mid b, c \,]| \; b \rightarrow c \rightarrow \text{STOP})$

$= a \rightarrow b \rightarrow (\text{STOP} \; |[\, a, b \mid b, c \,]| \; c \rightarrow \text{STOP})$

$= a \rightarrow b \rightarrow c \rightarrow \text{STOP}$

$traces = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, c\rangle\}$

Abaixo, um exemplo de *traces* de uma recursão:

$LIGHT = on \rightarrow off \rightarrow LIGHT$

A função de recursão é definida por $F(Y) = on \rightarrow off \rightarrow Y$.

$traces(\text{STOP}) = \{\langle\rangle\}$

$traces(F(\text{STOP})) = \{\langle\rangle, \langle on\rangle, \langle on, off\rangle\}$

$traces(F(F(\text{STOP}))) = \{\langle\rangle, \langle on\rangle, \langle on, off\rangle, \langle on, off, on\rangle, \langle on, off, on, off\rangle\}$

Ou seja,

$$\begin{aligned} traces(F^n(\text{STOP})) = & \{\langle on, off\rangle^I \mid 0 \le i \le n\} \\ & \cup \\ & \{\langle on, off\rangle^I \frown \langle on\rangle \mid 0 \le i \le n\} \end{aligned}$$

2.3.4 Especificação e validação com *traces*

2.3.4.1 Orientada a propriedade

Sistemas são desenvolvidos para satisfazer um particular requisito. No modelo de *traces*, a especificação de um processo pode ser verificada, através da aceitação ou não do mesmo.

Se $S(tr)$ é o predicado de tr, P satisfaz $S(tr)$ se $S(tr)$ possui todos os *traces* de P.

$P\ sat\ S(tr) = \forall\ tr \in traces(P) \bullet S(tr)$

2.3.4.2 Orientada a processos

Um processo $P1$ atende às especificações de $P0$ se cada *trace* de $P1$ é permitida por $P0$. Ou seja, $P1$ é considerado um refinamento de $P0$. A sua definição é dada por:

$P0 \sqsubseteq_T P1 = traces(P1) \subseteq traces(P0)$

2.3.5 *Stable Failures*

O modelo de *traces* está preocupado somente com as seqüências de eventos que um processo pode executar. Observar um processo envolve a análise de eventos à medida que eles ocorrem. Esta visão é apropriada para a análise de segurança, já que os *traces* associados com os processos possuem informações suficientes para a verificação de segurança.

A propriedade de *Liveness* por sua vez, está preocupada com o comportamento de um processo, ou seja, o quanto um processo permanece sem apresentar divergências ou estados indesejáveis. A divergência seria uma propriedade indesejada ao sistema, uma vez que eventos internos e imperceptíveis ao ambiente poderiam executar indefinidamente em um sistema, consumindo recursos. Com uma visão de processo como componentes interativos, um processo isolado pode nunca garantir que um determinado evento possa ocorrer, já que o sistema pode prevenir que o mesmo possa ocorrer.

2.3.5.1 Observando Processos

Um processo P que não pode efetuar nenhum processo interno é chamado de *stable*, escrito como $P \downarrow$:

$$P \downarrow = \neg\ (P \xrightarrow{\tau}\ )$$

Se para um dado conjunto $X \subseteq \Sigma^{\checkmark}$, caso não exista nenhum $a \in X$ que P possa executar, então P recusa X.

$$P\ ref\ X = \exists\ P' \overset{\langle\rangle}{\Rightarrow} P' \wedge P' \downarrow \wedge\ \forall\ a \in X \bullet \neg\ (P' \xrightarrow{a}\ )$$

Outra possibilidade seria caso P executasse eventos internos para sempre, nunca alcançando um estado estável. Neste caso, P é dito divergente, escrito como $P \uparrow$.

$$P \uparrow = \exists (P_i)_{i \in N} \bullet (P = P_0 \wedge \forall\ i \bullet P_i \xrightarrow{\tau} P_{i+1})$$

É possível que em um determinado ponto durante a execução de um conjunto X oferecido, um *trace* seja recusado pelo processo P. Esta recusa será guardada junto com sua seqüência finita de eventos tr ocorridas até o momento em que o conjunto X foi recusado. A observação (tr, X) é chamada de *stable failures* de P, ou seja:

$$\exists\ P'' \bullet P \overset{tr}{\Rightarrow} P'' \wedge P'' \downarrow \wedge\ P''\ \mathrm{ref}\ X$$

O processo pode executar os eventos em tr, e então alcançar um estado *stable* no qual ela recusa todos os eventos contidos em X, ocasionado um *deadlock*.

2.3.6 *Acceptances*

O conceito e semântica de *Acceptances* foram introduzidos na semântica $\mathcal{CSPP}$ (LAWRENCE, 2003b, 2002, 2003a, 2004)

A semântica *Acceptances* é baseada na idéia de oferecer a um processo eventos mutualmente exclusivos simultaneamente e observar qual evento pode ser escolhido ou *aceito*. Um processo P do tipo $a \rightarrow \text{STOP}$ é um processo em que apenas aceita o evento a, executando nada em seguida. Caso um conjunto de eventos $\{b, c\}$ seja oferecido a este processo, o mesmo não pode escolher nenhum desses dois eventos, ou seja, a saída é o conjunto vazio $\varnothing$. No caso, os eventos $\{b, c\}$ são recusados pelo processo P. Caso um conjunto de eventos $\{a, c\}$ seja oferecido a este mesmo processo P, o evento a será escolhido. Ou seja, $\{b, c\} \rightsquigarrow \varnothing$ e $\{a, c\} \rightsquigarrow a$, onde $\rightsquigarrow$ significa 'aceita' ou 'responde'.

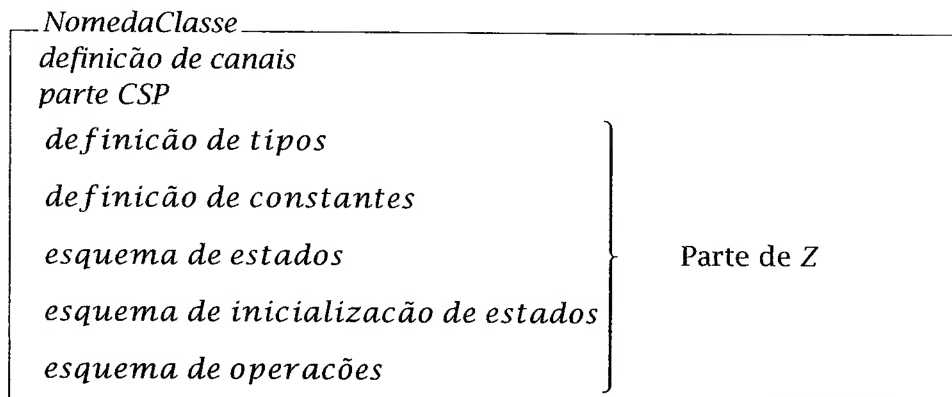
Este simples exemplo demonstra a idéia básica em relação ao CSP clássico. Um processo P do tipo $(a \rightarrow \text{STOP}) \;\Box\; (b \rightarrow \text{STOP})$, ou seja, um processo no qual pode-se executar tanto um evento do tipo a e parar, como um evento do tipo b e parar com

uma escolha externa determinada pelo ambiente externo. Se o ambiente externo oferecer a, então o processo executa a primeira opção, a segunda opção ocorre caso o evento b seja oferecido. Portanto $\{a\} \rightsquigarrow \{a\}$ e $\{b\} \rightsquigarrow \{b\}$. Caso o conjunto de eventos a, b sejam oferecido, o processo P pode tanto favorecer o evento a como o evento b, ou seja, $\{a, b\} \rightsquigarrow \{a, b\}$, o aceite não impõe uma escolha.

A semântica de *Acceptances* se adequa quando existe a ocorrência de prioridades, uma vez que caso o processo P fosse do tipo $(a \rightarrow \text{STOP})\overleftarrow{\Box}(b \rightarrow \text{STOP})$, caso os eventos a, b sejam oferecidos, o processo P favorece o evento a, ou seja, $\{a, b\} \rightsquigarrow \{a\}$.

2.4 Combinando *Object-Z* com CSP

Diversos trabalhos têm sido propostos para se integrar a notação CSP com a *Object-Z* (FISCHER; WEHRHEIM, 2004; FISCHER, 1997; SMITH; DERRICK, 2001; DERRICK; SMITH, 2003), além de diversos estudos envolvendo a notação Z com CSP (MOTA; SAMPAIO, 1998; MOTA; BORBA; SAMPAIO, 2002; MOTA; SAMPAIO, 2001). A combinação da versão orientada a objetos possui a seguinte estrutura:



Basicamente, a utilização da combinação destas duas linguagens compreende, de acordo com Derrick e Smith (2003), três fases distintas, em que a primeira fase envolve a especificação utilizando *Object-Z*, a segunda envolve modificações das interfaces das classes de modo a implementar as sincronizações e comunicações desejadas e, por fim, a terceira fase envolve a utilização de operadores CSP, definindo-se deste modo o sistema final.

A especificação de um sistema utilizando-se a combinação de *Object-Z* e CSP não necessita da utilização de todos os operadores utilizados em *Object-Z*, tanto como de CSP, ou seja, não é necessário, por exemplo, utilizarmos o operador de paralelismo definido em *Object-Z* uma vez que uma função similar é definida em CSP, bem como a notação de composição seqüencial definida em CSP.

2.4.1 *Failures* e *Divergencies*

A combinação de *Object-Z* com CSP é feita basicamente através da introdução da semântica de *failures-divergences* utilizadas em processos CSP às classes *Object-Z*. A semântica *failures-divergences* modela os processos através da tripla $(\mathcal{A}, \mathcal{F}, \mathcal{D})$, onde $\mathcal{A}$ é o alfabeto de eventos que um processo pode alcançar ou gerar, $\mathcal{F}$ são as falhas especificadas e $\mathcal{D}$ as suas divergências.

De acordo com a definição de Fischer (1997):

- O conjunto $\mathcal{F}$ é definido como $\mathcal{F} : \text{seq}\,\mathcal{A} \leftrightarrow \mathbb{P}\,\mathcal{A}$, ou seja, uma relação de *traces* em $\mathcal{A}$ e subconjuntos de $\mathcal{A}$. O par *(tr, X)* pertence a $\mathcal{F}$ se *tr* é uma seqüência finita de eventos possível de se executar e *X* é o conjunto de eventos que um sistema pode recusar logo após a execução de *tr*.
- O conjunto de *traces* $\mathcal{D}$ é definido como $\mathcal{D} : \mathbb{P}\,\text{seq}\,\mathcal{A}$ com $\mathcal{D} \subseteq \text{dom}\,\mathcal{F}$, no qual após a sua execução o sistema pode divergir.

As falhas definidas para as classes *Object-Z* podem ser definidas através do histórico de suas operações, definidas em (SMITH; DERRICK, 2001). O histórico de uma classe é um conjunto não vazio de seqüências de estados *S* combinado com uma seqüência de operações *O* que podem ser representados por:

$H \subseteq S^{\omega} \times O^{\omega}$

onde S^{ω} representa a seqüência possivelmente infinita de elementos do conjunto *S*, uma definição formal completa do modelo de histórico se encontra em Smith (1995).

Dado uma classe *C* e seu respectivo histórico *H*, *(t, X)* é uma falha de *C* se a seqüência finita de histórico *H* satisfaz as seguintes condições:

- A seqüência de operações do histórico corresponde à seqüência de eventos em *t*.
- Para cada evento *e* em *X*:
  - Não existe um histórico derivado do histórico original através de uma operação efetuada pelo evento *e*.
  - Existe um histórico derivado do histórico original através de uma operação efetuada por um segundo evento com uma mesma assinatura e valores de entrada de *e* e cujo evento não pertence a *X*.

A segunda condição implica que dado um conjunto *X*, as saídas de uma operação não podem ser definidas pelo ambiente, uma instância de uma classe pode recusar

todos os eventos com exceção de um evento em que a sua saída corresponde a uma operação particular e seus valores de entrada.

Definindo-se formalmente:

$$
\begin{array}{l}
failures(C) = \\
\quad \{(t, X) \mid \exists (s, o) \in H \bullet \\
\qquad\qquad s \in S^* \wedge t = events(o) \wedge \\
\qquad\qquad (\forall e \in X \bullet \\
\qquad\qquad\quad (\nexists st \in S, op \in O \bullet \\
\qquad\qquad\qquad e = event(op) \wedge (s \frown \langle st \rangle, o \frown \langle op \rangle) \in H) \\
\qquad\qquad\quad \vee \\
\qquad\qquad\quad (\exists (n, p) \in O, (n, q) \in O \bullet \\
\qquad\qquad\qquad \iota(p) = \iota(q) \wedge e = event((n, p)) \wedge \\
\qquad\qquad\qquad (\exists st \in S \bullet \\
\qquad\qquad\qquad\quad (s \frown \langle st \rangle, o \frown \langle (n, q) \rangle) \in H) \wedge \\
\qquad\qquad\qquad\quad event(n, q) \notin X))\}
\end{array}
$$

Onde a função *events* retorna a seqüência de eventos associados com a seqüência de operações e a função ι retorna a atribuição de valores aos parâmetros de entrada de uma operação.

Uma vez que *Object-Z* não permite "esconder"operações, a ocorrência de divergência não é possível, pois como toda operação efetuada pelo objeto é visualizável pelo meio externo, o estado do sistema é sempre determinístico. Portanto o conjunto de divergências de C_i é definido como:

$$divergences(C_i) = \varnothing$$

As divergências poderiam ser utilizadas para indicar um comportamento caótico que poderia ocorrer quando uma operação é efetuada fora de suas pré-condições, porém, de acordo com as definições da semântica de *Object-Z*, as operações que não se enquadram dentro das suas pré-condições são bloqueadas.

Apesar das divergências não ocorrerem em *Object-Z*, elas podem ocorrer na linguagem de especificação CSP através da utilização do operador de "*hidding*", gerando desta maneira um não determinismo ilimitado. Normalmente o não determinismo ocorre uma vez que um processo é caracterizado por um conjunto finito de *traces*, a possibilidade de um processo escolher entre uma quantidade infinita de opções pode levar a divergências já que é impossível de se distinguir se um processo escolhe uma seqüência finita de um evento a ou uma seqüência infinita de a's.

Para solucionar esse problema de não-determinismo ilimitado, foi utilizada a solução adotada em Smith e Derrick (2001), para se adequar o problema existente

na integração de *Object-Z* com CSP em que a operação de "hiding"é limitada por:

$$\forall s \in \text{dom} F \bullet \neg (\forall n \in \mathbb{N} \bullet \exists t \in C^* \bullet \#t > n \wedge s \frown t \in \text{dom} F)$$

Ou sejam, caso exista a possibilidade de ocorrência de seqüências ilimitadas, por convenção, as mesmas não são "escondidas"do ambiente externo que a controla.

2.4.2 Objetos ativos e objetos passivos

Objetos ativos são objetos que possuem o controle do processamento, enquanto que objetos passivos são objetos controlados por outros objetos em um sistema (MAHONY; DONG, 1998, 2000, 2002). Em TCOZ, a definição do método MAIN é utilizada para determinar o comportamento de objetos ativos de uma classe, logo após a sua inicialização.

Objetos ativos possuem também a característica de não compartilharem nenhum atributo de estados nem métodos locais com nenhum objeto, inclusive objetos pertencentes à mesma classe. Somente atributos constantes podem ser compartilhados entre classes. Todas as interações efetuadas com objetos ativos são efetuadas através da utilização de canais.

Objetos passivos, por sua vez, compartilham todos os métodos e variáveis com outros objetos que os controlam, exceto as variáveis ou os métodos que são "escondidos"do ambiente externo.

2.4.3 Comunicação externa com o ambiente

Na abordagem utilizada em Fischer (1997) e Mahony e Dong (2000) para se definir a interface entre as classes são utilizadas canais (*channel*). Apesar da abordagem efetuada por Mahony e Dong (2000) definir o canal como tipos heterogêneos, que aceita comunicar qualquer tipo de variável, a abordagem utilizada neste trabalho, por convenção, segue a linha definida em Fischer (1997), na qual toda declaração de canais utiliza a forma **channel** $c : [p_1;\ ty_1;\ ...;\ p_n;\ ty_n]$, onde c é o nome do canal e $p_1, ..., p_n$ são os nomes dos parâmetros e $ty_1, ..., ty_n$ a declaração dos tipos dos parâmetros.

Um canal c pode aparecer sozinho em uma declaração ou aplicada a valores v, utilizados em Z (SPIVEY, 1992) como $c.v$. A utilização convencional mais utilizada e adotada neste trabalho é $c?v$ (para valores v de entrada) e $c!v$ (para valores v de saída). Ao contrário de variáveis de estado, que são encapsuladas pelas classes, os canais são utilizados como variáveis compartilhadas entre objetos, reduzindo-se desta maneira a utilização de referências entre as classes e aumentando-se a modularidade do sistema.

A utilização de canais especiais em que são descritos sensores e atuadores pode ser vista em Mahony e Dong (2002, 1999, 1998) **sensor** e **actuator**, sendo **sensor** descrito basicamente como canais de leitura e **actuator** como canais de escrita, utilizando-se, neste caso, a notação temporal de CSP (TCSP), definida em Schneider (2000). Uma das principais diferenças existentes entre as várias propostas de se integrar *Object-Z*, CSP e questões temporais é a modelagem em relação a variáveis de escrita e leitura em relação ao ambiente externo. Em Smith e Hayes (2000, 2002), as leituras e escritas podem ser tratadas tanto como variáveis discretas ou variáveis contínuas. Por exemplo, a seguinte anotação $v : \mathbb{T} \rightarrow \mathbb{R}$ é equivalente à sua versão discretizada de $v : \mathbb{T} \rightsquigarrow \mathbb{R}$ com tempo de discretização de 0.1 segundos.

loc_a

$la, ra : \textbf{chan}$
$a : A$

$\textsc{Main} \mathrel{\widehat{=}} \mu LA \bullet ([i : A] \bullet la?i \rightarrow a := i;\ LA) \,\square\, (ra!a \rightarrow LA)$

Tal anotação é efetiva somente para sistemas fechados. Ou seja, processos modelados utilizando tais canais como interface não podem ser entendidos isoladamente, levando a um sistema altamente acoplado. Em contraste o mecanismo de **sensor** e **actuator** fornece um mecanismo localizado de interfaces assíncronas, facilitando bastante o seu projeto e entendimento, uma vez que isola as particularidades do ambiente.

A utilização de **sensor** e **actuator** também possibilita que as classes de objetos possam utilizar interfaces relacionadas a estas funções contínuas de maneira transparente.

2.5 Exemplo: Produtor e Consumidor

Neste seção, através de um exemplo, demonstraremos os passos utilizados para o desenvolvimento de um sistema, por meio de uma especificação inicial utilizando-se a linguagem de especificação CSP-OZ e, a partir da sua especificação, transformá-la em uma implementação em Java.

O sistema produtor-consumidor é um sistema simples, composto de um processo produtor que introduz itens em um sistema de armazenamento e por um processo consumidor que retira itens deste mesmo sistema de armazenamento. As características deste sistema seguem regras pré-determinadas, como por exemplo o limite de itens que o sistema de armazenamento pode suportar, as prioridades entre os processos consumidor e produtor e a política de retirada de elementos

(por exemplo, uma política na qual os primeiros itens a entrar no sistema são os primeiros a serem retirados).

2.5.1 Descrição do exemplo

Apesar da linguagem Java permitir a implementação de sistemas concorrentes, sua biblioteca padrão é baseada na utilização de monitores, *threads* e semáforos. Para a utilização de operadores CSP existem disponíveis duas bibliotecas com modelos de processos e canais JCSP (P.D.AUSTIN; P.H.WELCH, 2000) e CTJ (HILDERINK, 2000). Apesar de pequenas diferenças existentes entre estas duas bibliotecas (SCHALLER; HILDERINK; WELCH, 2000), ambas atendem os requisitos propostos para se implementar a semântica definida por CSP. Para este presente trabalho, a biblioteca escolhida foi a JCSP.

Existem diversos trabalhos propostos para a transformação de linguagens de especificação baseada nas combinações de CSP (HOARE, 1985) e *Object-Z* (SMITH, 1995) para linguagens de programação em Java utilizando-se as bibliotecas JCSP e CTJ, podendo-se listar, por exemplo, os trabalhos propostos em Raju, Rong e Stiles (2003), P.H.Welch e J.M.R.Martin (2000), Nevison (2001) e Cavalcanti e Sampaio (2002).

A utilização da biblioteca JCSP é exemplificada pelo modelo de processos produtor e consumidor, uma versão de um exemplo similar utilizando-se a biblioteca CTJ pode ser encontrada em Cavalcanti e Sampaio (2002).

2.5.2 Biblioteca JCSP

Java Communicating Sequential Processes (JCSP) (P.D.AUSTIN; P.H.WELCH, 2000) é uma biblioteca Java que implementa os operadores CSP como canais, paralelismo, guardas, etc, além da implementação de primitivas da linguagem OCCAM (HOARE, 1988) (por exemplo, operadores relacionados à prioridade de escolha entre diversos guardas).

A utilização de programação concorrente em Java é baseada em *Threads*, o seu controle é baseado através da utilização direta de métodos de uma classe. O controle de uma *Thread* é feito através do uso adequado de diversos métodos disponíveis, porém, a sincronização entre diversas *Threads* é muito complicada, introduzindo uma complexidade maior ao sistema. Esta complexidade é aumentada quando múltiplas *Threads* acessam uma mesma variável compartilhada.

Em contrapartida, JCSP implementa uma biblioteca simples que implementa o conceito de processo. Um processo possui apenas uma linha de execução e a co-

municação interprocessos é efetuada através de canais, ou seja, através do envio e recebimento de mensagens. Os processos não invocam métodos de outros processos, podendo executar seqüencialmente ou paralelamente com outros processos. Um processo pode também esperar passivamente através da chegada de eventos, e a sua execução é feita através do disparo de algum evento externo, estes mesmos eventos podem possuir prioridades sobre outros ou funcionar de modo que todos os eventos tenham prioridades iguais ou através de alguma regra arbitrária.

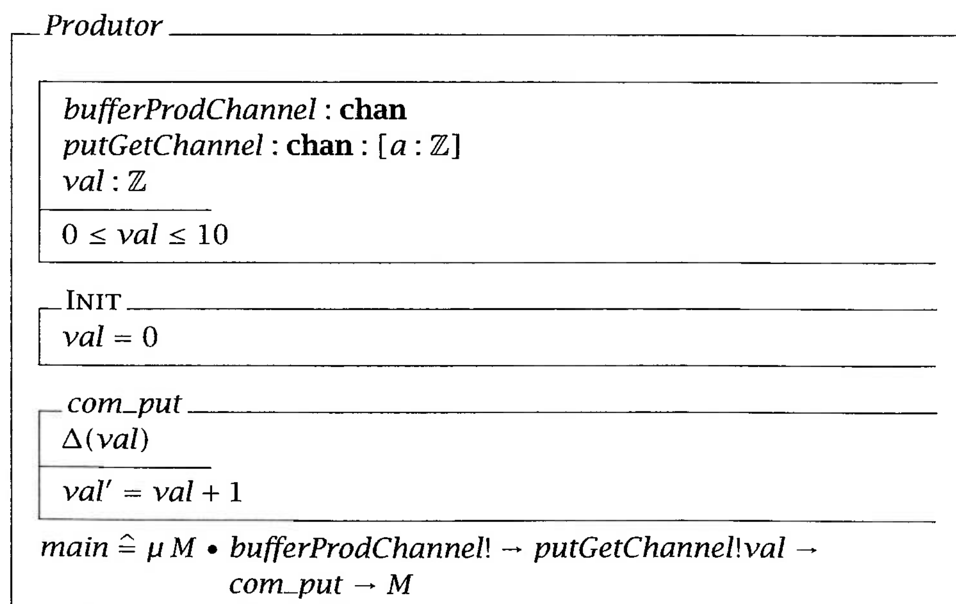
Uma outra biblioteca desenvolvida em Java que implementa operadores CSP é CTJ (*"Communicating Threads for Java"*) (HILDERINK, 2000). As principais diferenças entre JCSP e CSP podem ser encontradas em Schaller, Hilderink e Welch (2000).

2.5.3 Especificação

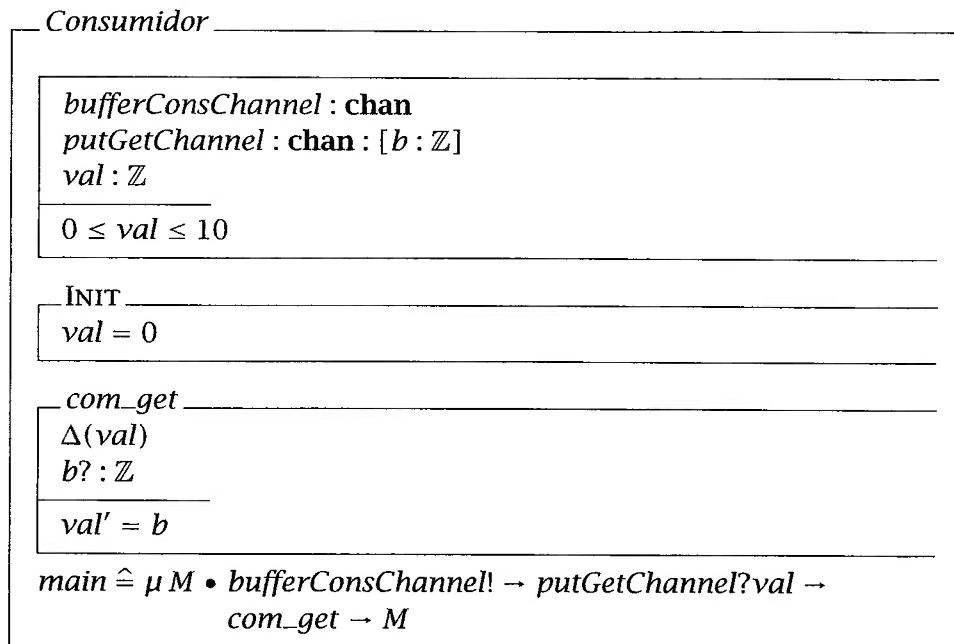
Para se criar o exemplo produtor-consumidor, foram criadas 4 classes:

- A classe *Produtor*, responsável pela produção de itens à classe *Buffer*.
- A classe *Consumidor*, responsável pelo consumo de itens à classe *Buffer*.
- A classe *Buffer*, responsável pelo armazenamento e retirada de itens, inseridas e retiradas respectivamente pela classe *Produtor* e *Consumidor*.
- A classe *Main*, que instancia as classes *Produtor*, *Consumidor* e *Buffer*.

Tanto a classe *Produtor* como a classe *Consumidor* são sincronizadas com a classe *Buffer* através dos canais *bufferProdChannel* e *bufferConsChannel* respectivamente. Após a sincronização dos mesmos, os dados são inseridos ou retirados através do canal *putGetChannel*.



Pode-se perceber que os dados inseridos e lidos pela classe *Buffer* estão definidos entre 0 e 10.


Consumidor

$bufferConsChannel : \mathbf{chan}$
$putGetChannel : \mathbf{chan} : [b : \mathbb{Z}]$
$val : \mathbb{Z}$

$0 \leq val \leq 10$

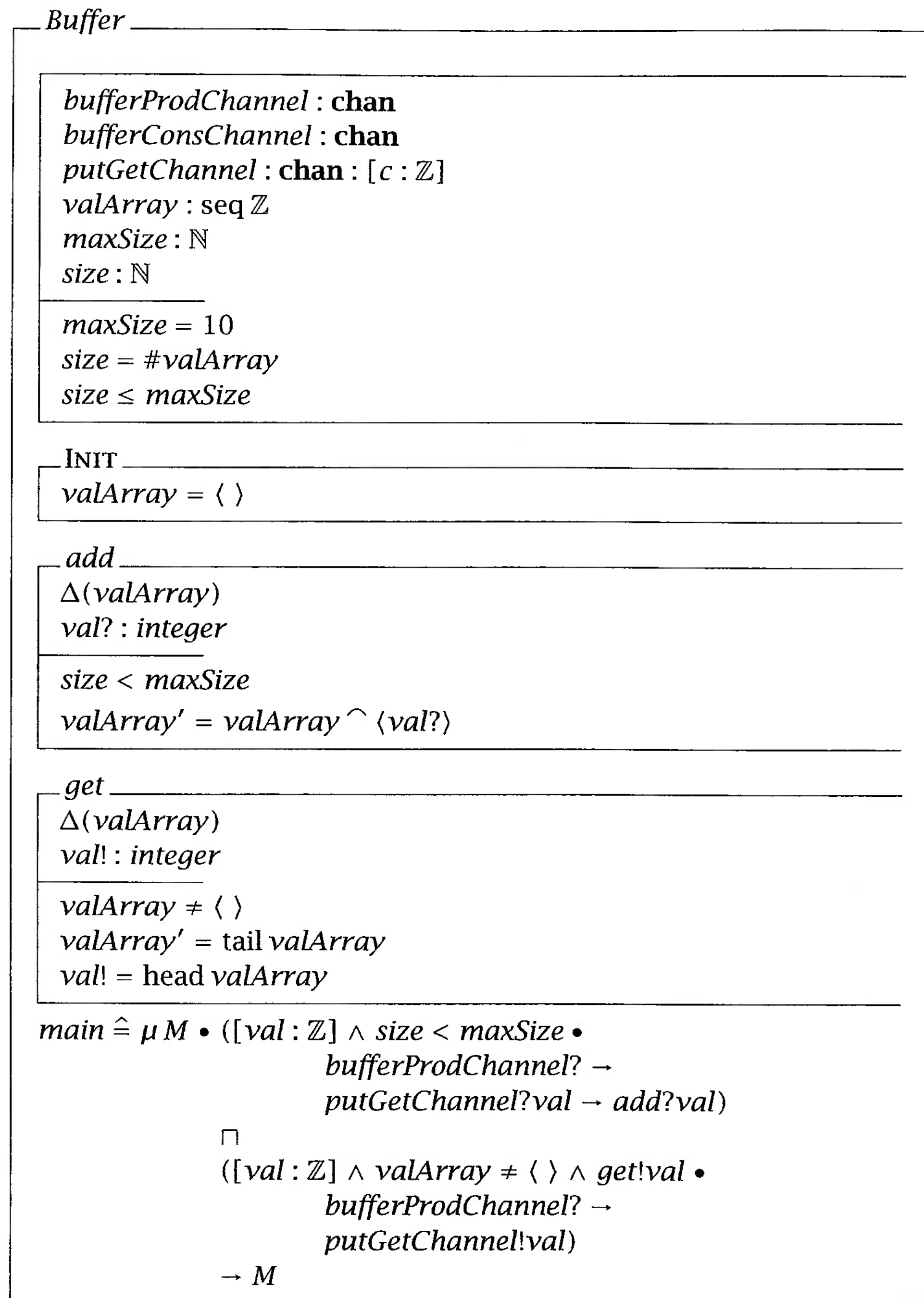
INIT
$val = 0$

com_get
$\Delta(val)$
$b? : \mathbb{Z}$

$val' = b$

$main \mathrel{\widehat=} \mu M \bullet bufferConsChannel! \rightarrow putGetChannel?val \rightarrow com_get \rightarrow M$

Para implementação da classe *Buffer*, é bom salientar que a sincronização entre as classes *Produtor* e *Consumidor* é feita de maneira condicional, ou seja, a classe *Consumidor* pode apenas sincronizar caso a classe *Buffer* possua algum elemento e a classe *Produtor* pode sincronizar caso o limite de elementos armazenados na classe *Buffer* não tenha atingido.



```
Buffer
    bufferProdChannel : chan
    bufferConsChannel : chan
    putGetChannel : chan : [c : ℤ]
    valArray : seq ℤ
    maxSize : ℕ
    size : ℕ
    ──────────
    maxSize = 10
    size = #valArray
    size ≤ maxSize

    INIT
    valArray = ⟨ ⟩

    add
    Δ(valArray)
    val? : integer
    ──────────
    size < maxSize
    valArray′ = valArray ⁀ ⟨val?⟩

    get
    Δ(valArray)
    val! : integer
    ──────────
    valArray ≠ ⟨ ⟩
    valArray′ = tail valArray
    val! = head valArray

    main ≙ μ M • ([val : ℤ] ∧ size < maxSize •
                      bufferProdChannel? →
                      putGetChannel?val → add?val)
                 ⊓
                 ([val : ℤ] ∧ valArray ≠ ⟨ ⟩ ∧ get!val •
                      bufferProdChannel? →
                      putGetChannel!val)
                 → M
```

A classe *Buffer* pode ser representada pelo operador seq, correspondente a uma seqüência de inteiros, a variável *size*, representa o tamanho de *valArray* através da notação #, sendo por definição, menor do que o valor *maxSize*. O método *add* adiciona valores inteiros caso o *valArray* não tenha atingido o valor limite (*maxSize*) e o método *get* retira valores inteiros de *valArray* caso a mesma não esteja vazia. O operador tail corresponde a todos os valores da seqüência de inteiros com exceção do primeiro valor, o operador head corresponde ao primeiro elemento da seqüência.

O operador ⊓ é relacionado à escolha interna efetuada pelo processo *Buffer* em relação à inserção e retirada de elementos caso as condições de processamento estejam habilitadas e os processos *Produtor* e *Consumidor* estejam habilitados. A leitura e escrita nos canais são descritas através dos operadores ? e ! respectiva-

mente.

A classe principal, por sua vez, pode ser definida como uma composição paralela dos três processos (*Produtor*, *Consumidor* e *Buffer*).

$$\begin{aligned} Main \mathrel{\hat{=}} & (Produtor \mid[\ putGetChannel\ ]\mid Consumidor) \\ & \mid[\ bufferProdChannel, bufferConsChannel, putGetChannel\ ]\mid \\ & Buffer \end{aligned}$$

A transformação da classe *Produtor* do exemplo acima para a linguagem JAVA é exemplificada abaixo. A comunicação entre a classe *Produtor* e a classe *Buffer* é efetuada através da sincronização do canal *bufferChannel*. Após a sua sincronização, o valor da classe é armazenado para a utilização da classe *Consumidor*.

```
while(true){
    bufferChannel.write(null);
    putChannel.write(val);
    val++;
    val = val % 10;
}
```

Analogamente à classe *Consumidor*, o programa JAVA ficaria:

```
while(true){
    bufferChannel.write(null);
    val = getChannel.read();
}
```

Já a classe *Buffer*, além dos canais de sincronização previamente definidos, utiliza-se de uma sincronização condicional através da classe *Alternative* e *Guard*. A classe *Alternative* possibilita ao processo esperar passivamente até que algum dos eventos relacionados à classe *Guard* seja habilitado. No exemplo acima, além da utilização da classe *Guard*, uma pré-condição é imposta à mesma, no caso, o tamanho limite da classe *Buffer* e a necessidade de pelo menos um elemento na classe *Buffer* para que a sincronização com as classes *Produtor* e *Consumidor* seja habilitada.

O processo da classe *Buffer* ficaria desta maneira implementada como:

```
final Guard[] guards = {bufferProdutorChannel,
                        bufferConsumidorChannel};
final boolean[] preCondition = new boolean[guards.length];
final int PRODUDOR = 0;
final int CONSUMIDOR = 1;
Alternative alt = new Alternative (guards);
```

```
        int auxInt;
        while(true){
            preCondition[PRODUDOR] =
                                (intVector.size() <= LIMIT_BUFFER);
            preCondition[CONSUMIDOR] = (intVector.size() > 0);
            switch (alt.priSelect (preCondition)) {
                case(PRODUDOR):
                    bufferProdutorChannel.read();
                    auxInt = putGetChannel.read();
                    intVector.add(new Integer(auxInt));
                    break;
                case(CONSUMIDOR):
                    bufferConsumidorChannel.read();
                    auxInt = ((Integer)intVector.get(0)).
                                  intValue();
                    putGetChannel.write(auxInt);
                    intVector.remove(0);
                    break;
            }
        }
```

Já a classe principal seria responsável pela criação dos canais e a instanciação dos mesmos. A sua implementação pode ser exemplificada logo abaixo, com a utilização da classe *Parallel* e os canais *One2OneChannel* e *One2OneChannelInt*:

```
    One2OneChannel bufferProdutorChannel = new One2OneChannel();
    One2OneChannel bufferConsumidorChannel = new One2OneChannel();
    One2OneChannelInt putGetChannel = new One2OneChannelInt();
    Buffer buffer = new Buffer(bufferProdutorChannel,
                               bufferConsumidorChannel,
                               putGetChannel);
    Produtor produtor = new Produtor(bufferProdutorChannel,
                                     putGetChannel);
    Consumidor consumidor = new Consumidor(
                                     bufferConsumidorChannel,
                                     putGetChannel);
    new Parallel (
        new CSProcess[] {
                buffer,
                produtor,
                consumidor
        }
    ).run ();
```

O diagrama de classes do sistema produtor-consumidor pode ser visualizado na figura 2.1:

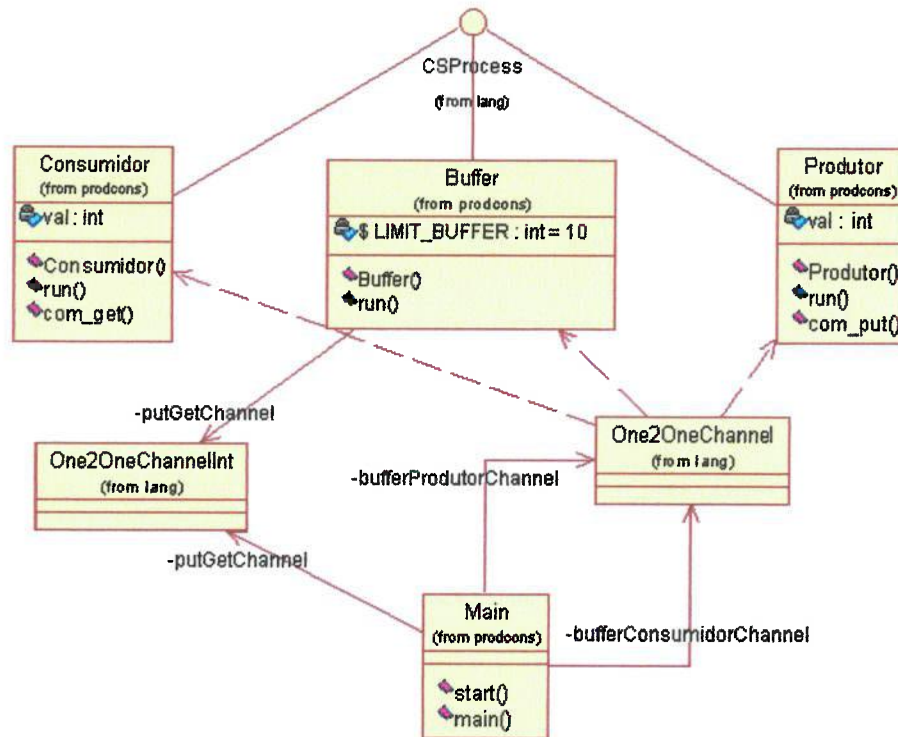


Figura 2.1: Diagrama de classes do sistema produtor-consumidor.

Pode-se perceber que as classes *Produtor*, *Consumidor* e *Buffer* implementam a interface CSProcess, além da utilização dos canais *One2OneChannel* e *One2OneChannelInt*.

3 Padrões de projeto

Este capítulo tem por finalidade descrever as principais características de padrões de projeto, as principais considerações efetuadas pelo clássico livro de Gamma, Helm e Johnson (1995) e os padrões de projeto utilizados em sistemas de tempo real. A primeira seção é uma introdução ao conceito de padrões de projeto, a segunda descreve as principais considerações sobre padrões de projeto utilizados em sistemas de tempo real e adotados por este trabalho.

Na subseção 3.2.1 são descritas as principais características dos padrões de distribuição de recursos. A subseção 3.2.3 descreve as características do padrão de escalonamento, bem como a utilização de operadores CSP na descrição e formalização de um modelo de escalonador. A subseção 3.2.6 descreve as necessidades dos padrões de segurança e analisa as características da implementação de um *Watchdog*, bem como o inter-relacionamento deste padrão de projeto em relação a outros subsistemas. A subseção 3.2.8 descreve as características relacionados ao tratamento de eventos assíncronos que ocorrem em um sistema, normalmente ocorridos após uma execução externa do sistema através de uma interface homem-máquina.

Os padrões de projeto para sistemas de tempo real apresentados nas subseções abaixo são especificados através da utilização da linguagem de especificação CSP-OZ.

3.1 Introdução

Desenvolvedores experientes, ao tentar solucionar um novo problema, normalmente percebem que a solução para este problema possui uma solução bastante comum a uma previamente criada ou vista. Os problemas podem não ser exatamente iguais, e uma solução idêntica dificilmente os resolveria, mas devido à semelhança entre os mesmos, uma solução semelhante ou uma leve adaptação da solução muitas vezes resolve este novo problema. Esta solução genérica para um problema recorrente é chamada de padrão de projeto. A criação de um padrão de projeto envolve a observação de similaridades entre dois problemas e a sua solução correspondente de modo que os aspectos genéricos e comuns da solução original possam ser aplicados

a este novo problema (GAMMA; HELM; JOHNSON, 1995; SHALLOWAY, 2004).

Apesar da utilização, desenvolvimento e definições de padrões em qualquer tipo de atividade técnica serem largamente utilizados há anos, os estudos e as definições de padrões de projeto para o desenvolvimento de software são encontrados em larga escala somente no início da década de 90, quando as utilizações de linguagens orientadas a objetos e metodologias orientadas a objetos passaram a ser amplamente empregadas na indústria.

A aplicação de padrões de projeto envolve basicamente dois fundamentos:

- O primeiro fundamento relacionado a padrões de projeto envolve a aplicação dos padrões, ou seja, a princípio são identificados os problemas e subseqüentemente examinados os padrões de soluções existentes. Dentre as soluções identificadas, são definidas aquelas que melhor atendam à solução do seu problema.
- O segundo fundamento é a identificação e captura de novos padrões para serem adicionados a um banco de dados no qual o conhecimento é guardado para uma posterior consulta. Sua identificação envolve a abstração do problema para um nível de propriedades essenciais, criando uma solução genérica e entendendo as conseqüências da adoção desta solução no contexto do problema em que é aplicada.

A utilização de padrões não pode ser definida apenas como uma reutilização de software, e sim, uma reutilização de conceito. O padrão de projeto é uma otimização de um modelo de análise. O modelo de projeto difere do modelo de análise pelo fato do primeiro conter aspectos não necessários, mas incluídos de modo que faça o sistema funcionar de uma maneira particular e com efeitos particulares.

3.1.1 Importância na utilização de padrões de projeto

Os seguintes motivos podem ser enumerados para justificar a importância da utilização e dos estudos de padrões de projeto, segundo Shalloway (2004), Gamma, Helm e Johnson (1995).

- Reutilização de código.
- Estabelecer terminologias comuns para melhorar a comunicação entre diferentes times de projeto.
- Mudar o nível de desenvolvimento para uma perspectiva mais abstrata.
- Escolher a melhor solução possível, não somente aquela que funcione.

- Aumentar o aprendizado individual e o aprendizado da equipe.
- Aumentar a manutenibilidade do código.
- Facilitar a adoção de alternativas de padrões melhorados.

3.1.2 Classificação de padrões de projeto

A classificação de padrões de projeto, de acordo com Gamma, Helm e Johnson (1995) pode ser separada basicamente através da definição de duas principais características de um padrão:

- **Propósito** - A aplicabilidade do padrão de projeto.
- **Escopo** - Define basicamente se um padrão de projeto é aplicado a uma classe ou a um objeto.

Um padrão de classe trata da relação entre classes e suas subclasses, através da definição de cada uma das classes, definidas estaticamente. Padrões de objeto por sua vez, tratam das relações existentes entre objetos e que podem ser alteradas em tempo de execução, dando a elas uma maior flexibilidade.

Segundo o critério de aplicabilidade de padrões de projeto, estes podem ser classificados como:

- **Criacionais** - Relacionadas ao modo de criação de objetos.
- **Estruturais** - Relacionadas ao modo como são compostos classes ou objetos.
- **Comportamentais** - Relacionadas ao modo como classes ou objetos interagem entre si e a maneira como são distribuídas as responsabilidades

Em relação a padrões de projeto de sistemas de tempo real, a sua classificação, segundo Douglass (2002), pode ser classificada baseada em critérios arquiteturais de um sistema:

Para sistemas de tempo real, os padrões de arquitetura podem ser divididos em quatro aspectos primários, enumerados abaixo:

- Organização e arquitetura em larga escala.
- Gerenciamento de concorrência e recursos.
- Distribuição através de múltiplos espaços de endereços.
- Aspectos de segurança e confiabilidade.

Um modelo de sistema engloba todos os aspectos acima. O primeiro aspecto, relacionado à organização e arquitetura em larga escala, é relacionado à visão geral dos subsistemas e componentes. Ela organiza o sistema em alto nível, otimizando a maneira como cada uma das partes interagem entre si, como elas são construídas ou como elas são gerenciadas.

O segundo aspecto, o gerenciamento de concorrência e recursos, é relacionado ao modo como o sistema gerencia os processos e recursos finitos que a compõem, ou seja, um dos aspectos mais importantes tratando-se de sistemas de tempo real, uma vez que ela é responsável pela definição do determinismo e dos aspectos a serem otimizados para se alcançar os resultados desejados.

O terceiro aspecto é de como os objetos são distribuídos através de múltiplos espaços de endereços e, possivelmente, computadores remotos. Ela inclui as políticas e procedimentos para que cada um dos objetos procure o outro e cooperem entre si, incluindo-se neste aspecto, os protocolos de comunicação.

Por fim, o último aspecto trata dos problemas relacionados à segurança e confiabilidade, ou seja, o modo como o sistema se comporta com uma situação de erro ou falha.

3.1.3 Critérios de escolha de padrões de projeto

Para um sistema de tempo real, os critérios de escolha de um padrão de projeto levam em consideração aspectos necessários ao funcionamento do sistema. A escolha de um padrão de projeto deve ser criteriosamente definida de modo a que arquitetura atinja os requisitos do sistema. Abaixo, estão enumerados alguns tópicos de otimização de um modelo de análise descritos por Douglass (2002) para sistemas de tempo real.

- Performance
  - Pior Caso
  - Média
- Previsibilidade
- Saída do sistema
  - Média
  - Pico
- Confiabilidade
  - Tratamento de erros

- Tratamento de falhas

- Segurança
- Reusabilidade
- Distribuição dos recursos
- Portabilidade
- Manutenibilidade
- Escalabilidade
- Complexidade
- Utilização de recursos
  - Memória
  - Barramentos
- Consumo de energia
- Custos com hardware
- Custos de desenvolvimento

Tipicamente, a otimização de um sistema envolve simultaneamente mais de um tópico enumerado acima. Os requisitos de qualidade de serviço normalmente dirigem os aspectos de otimização do projeto, uma vez que eles definem os critérios de aceitação e a otimização mínima necessária. Para se otimizar todos os aspectos, são necessários ordenar cada um dos tópicos de acordo com a sua importância, com seus diferentes pesos para o sucesso do projeto ou produto. Em alguns sistemas, segurança e pior desempenho podem ser cruciais, enquanto que reusabilidade pode ser menos importante. Em outros sistemas, *time to market* (lançamento ao mercado) e portabilidade podem ser mais importantes do que questões relacionadas a desempenho. Portanto, um bom *design* otimiza o modelo de análise com todos os aspectos de qualidade de serviço, ordenados de acordo com a sua importância para o sucesso do projeto ou produto.

Uma vez que *design* está estritamente relacionado à otimização, os padrões de projeto de um sistema de tempo real também estão diretamente relacionados a otimização. Os padrões de projeto otimizam alguns aspectos do sistema, podendo de alguma maneira "piorar"outros aspectos. A definição de um conjunto de padrões de otimização é um dos tópicos mais importantes para se alcançar um bom balanceamento de modo a encontrar a melhor solução que atenda o problema.

Portanto, entendemos como padrão um conjunto de três aspectos. Primeiramente, o problema propriamente dito, segundo, a solução do problema, ou seja, o padrão a ser escolhido, e por fim as conseqüências geradas pela solução escolhida, uma vez que um padrão otimiza alguns aspectos em detrimento de outros, é necessário compreender tanto os aspectos positivos como os negativos do padrão.

Padrões de projeto podem ser aplicados a diferentes níveis de escopo. Padrões de arquitetura normalmente afetam a maioria dos sistemas, assim, padrões de arquitetura são estrategicamente aplicados a todos os sistemas.

3.2 Padrões de projeto para sistema de tempo real

Os padrões de projeto mais utilizados em sistemas de tempo real, em relação ao escalonamento de processos, alocação de recursos, segurança e entrada/saída de dados são analisados nas seções subseqüentes. Uma abordagem formal, utilizando-se CSP-OZ é efetuada de tal modo que podemos aplicar estes mesmos padrões a futuros sistemas de maneira simples e eficaz.

3.2.1 Padrões de Distribuição de Recursos

A distribuição de recursos é um aspecto importante em um sistema computacional, uma vez que ela define padrões de comunicação, acesso, procedimentos e estruturas para o acesso e a alteração de recursos existentes no sistema.

Os padrões de distribuição podem ser definidos em dois tipos:

- Assimétricos: Quando os recursos são conhecidos em tempo de *design*. Normalmente todos os sistemas de tempo real se enquadram neste tipo de padrão de distribuição devido à sua simplicidade.
- Simétricos: Quando os recursos não são conhecidos em tempo de *design* e a sua localização é definida em tempo de execução, utilizada principalmente quando um alto nível de flexibilidade é requerido.

Para a análise da distribuição de recursos, os protocolos envolvidos para a comunicação entre os diversos objetos e, conseqüentemente, as particularidades de cada um dos protocolos foram abstraídos. A análise é basicamente focalizada no nível da aplicação e arquitetura.

Dentre os diversos padrões de distribuição de recursos, podemos listar:

- *Shared Memory*: Onde a informação é armazenada em uma área comum de

memória, acessada por múltiplos processos. Normalmente definida ao nível de *hardware*.

- *Remote Method Call*: Onde a informação é obtida através de chamadas de sistema, utilizada em comunicação interprocesso.
- *Observer*: Basicamente é um mecanismo onde o padrão notifica os clientes sobre a modificação de alguma informação guardada. Este mecanismo serve de base para a implementação de outros padrões de distribuição como *Data Bus*, *Proxy* e *Broker*.
- *Data Bus*: Define um lugar comum para o compartilhamente de informações.
- *Proxy*: Basicamente é uma versão distribuída do padrão *Data Bus*, onde a escrita e leitura de informações são primeiramente gerenciadas por uma classe especial e posteriormente redirecionadas para o servidor real.
- *Broker*: Basicamente é uma versão simétrica do padrão *Proxy*, ou seja, a localização dos recursos não são conhecidas em tempo de *design*.

O padrão escolhido para análise e adotado por este trabalho foi o padrão *Data Bus* (DOUGLASS, 2002), uma vez que ela permite uma distribuição assimétrica, abstraindo-se também, particularidades de hardware e sistema operacional, além da sua simplicidade em relação ao padrão *Proxy*. Suas características são descritas logo abaixo.

3.2.2 Padrão *Data Bus* - Descrição

O padrão *Data Bus* define um lugar único para o compartilhamento de informações entre os processos. Processos clientes que desejam informações possuem um lugar comum onde podem inserir dados a serem compartilhados ou receber informações pertinentes. O padrão *Data Bus* é basicamente um padrão no qual é abstraída toda complexidade da localização dos dados através de um canal centralizado, em que vários objetos clientes podem estar conectados.

Muitos sistemas precisam compartilhar diferentes dados entre servidores e clientes, sendo que alguns sistemas podem não saber quando ou onde o cliente ou o dado foi implementado. Este padrão soluciona o problema através de um canal centralizado de dados onde os mesmos são compartilhados.

A estrutura do padrão é basicamente uma variação de escrita e leitura de dados. Na versão de leitura, o objeto cliente faz a leitura do novo dado de interesse através da requisição do dado à classe *DataBus* através da implementação da classe *Concrete Client*. Na versão de escrita, o objeto *Listener* efetua a escrita no *Data Bus*, que por sua vez notifica todos os objetos clientes interessados em saber da atualização do novo dado.

O padrão ainda utiliza descritivos a respeito de seus dados através de metadados, ou seja, informações a respeito do seu conteúdo, como tipo de dado, identificação e as unidades armazenadas. Os metadados são úteis uma vez que permitem a identificação dos dados de modo que os mesmos sejam utilizados adequadamente, mesmo que eles sejam armazenados em uma unidade diferente da desejada pelo objeto cliente. O servidor de metadados serve para desacoplar a implementação do cliente ao servidor.

O diagrama de classes da figura 3.1 exemplifica a estrutura básica das classes que compõem um padrão de projeto do tipo *DataBus*.

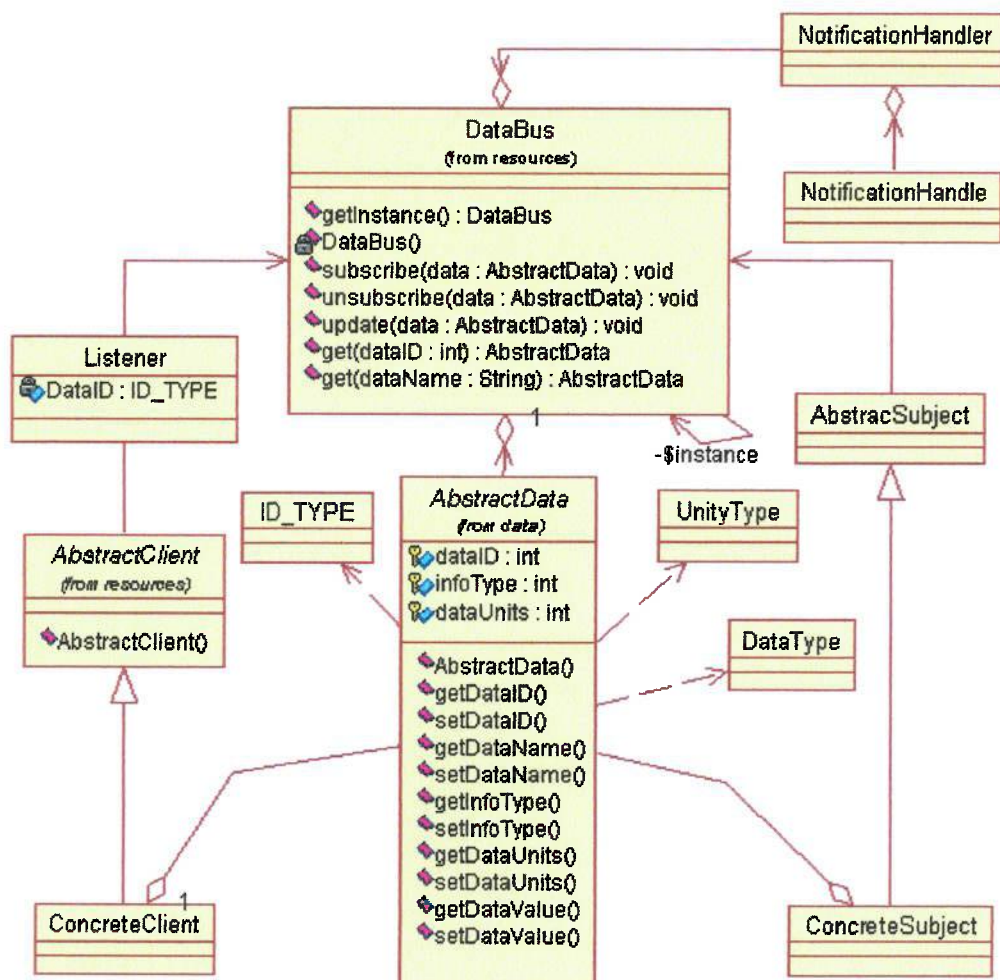


Figura 3.1: Diagrama de classes do padrão de projeto *DataBus*.

A classe *AbstractData* é responsável pela definição de todos os dados inseridos no *DataBus*, com a definição do identificador, o nome, o tipo, a unidade e o valor do dado.

3.2.2.1 Descritivo e formalização das classes

Para a utilização do padrão de projeto *DataBus*, definiremos os seguintes dados a serem armazenados na classe responsável pelo armazenamento centralizado de dados, em que os tipos de atributos utilizados dependem muito da necessidade de sua aplicação. Para a análise inicial, alguns tipos de dados básicos são definidos abaixo:

$$[STRING, INTEGER, LONG, FLOAT, DOUBLE, BOOLEAN]$$

Os tipos *ID_TYPE*, *DATA_TYPE* e *UNIT_TYPE* são tipos enumerados dependentes da aplicação e definidos conforme a sua necessidade. Os tipos *ACCESS_TYPE* por sua vez, descrevem a natureza de entrada e saída do dado e podem possuir os seguintes valores:

$$\begin{aligned} ACCESS_TYPE ::= &\ READ_TYPE \\ |&\ WRITE_TYPE \\ |&\ READ_WRITE_TYPE \end{aligned}$$

Ou seja, um dado armazenado pode ser utilizado somente para leitura, somente para escrita ou em ambas as situações.

O tipo *RATE_TYPE* define o tipo de tratamento utilizado pelos processos, em relação à leitura e escrita de dados. Dependendo do seu valor, as atualizações dos seus dados são efetuadas em uma taxa mais altas ou mais baixas, conforme a sua necessidade.

$$\begin{aligned} RATE_TYPE ::= &\ HIGH \\ |&\ MEDIUM \\ |&\ LOW \end{aligned}$$

Acima, são definidos apenas três níveis de taxa de atualização dos dados, sendo que conforme a necessidade, as suas taxas poderiam ser redefinidas.

As classes da figura 3.1 são descritas logo abaixo:

- *Abstract Data*

  A classe *Abstract Data* especifica a estrutura básica de todas os objetos de dados conectados ao barramento de dados. Os atributos importantes são:

  - *DataID*: *ID_TYPE* - Identificação do dado.
  - *DataName*: *String* - O nome do dado disponibiliza uma alternativa para a identificação do dado.
  - *InfoType*: *DATA_TYPE* - O tipo de dado.

- *DataUnit*: *UNIT_TYPE* - Este atributo identifica a unidade referente ao dado, por exemplo, unidades de peso, comprimento, etc.
- *Type*: ACCESS_TYPE - Atributo de identificação referente a leitura e escrita de dado.
- *DataRate*: *RATE_TYPE* - Atributo referente à taxa de atualização de leitura ou escrita de dado.

· *Concrete Data*

É uma subclasse da classe *Abstract Data* que define um dado específico do sistema, possuindo um *DataID* e um *DataName* específico. A subclasse ainda inclui um atributo referente ao seu valor, bem como a definição do tipo de variável em relação à sua escrita e leitura e sua taxa de atualização.

· *DATA_TYPE*

Enumeração de dados primitivos utilizados para representar valores na classe *Concrete Data*, como *INTEGER*, *LONG*, *FLOAT*, *DOUBLE*, *STRING*, etc. Este metadado permite registradores de manusear diferentes formatos de dados.

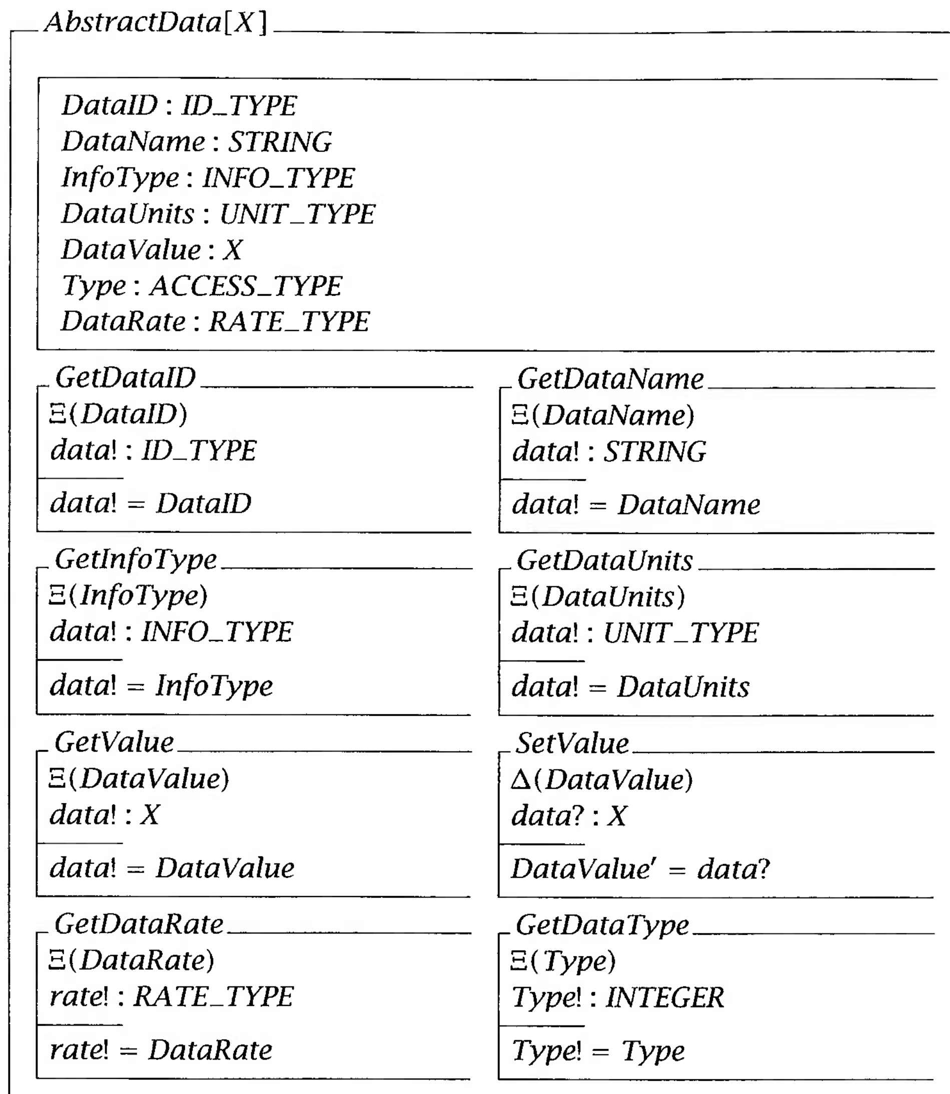
· *ID_TYPE*

O *ID_TYPE* permite identificar o nome lógico que representa o *Concrete Data* no sistema. Podem existir diversos objetos com o tipo *int*, mas o *ID_TYPE* permite que uma informação específica de um dado seja conhecida. Por exemplo: Velocidade = 0, SensorDePosicionamento = 1, PressaoDaValvula = 3, PesoDaEsteira = 4, EstadoDoChuveiro = 5.

· *UNIT_TYPE*

O *UNIT_TYPE* é uma enumeração de tipos apropriados de dados como, por exemplo, GRAMAS, KILOGRAMAS, METROS, GRAUS, MILHAS, etc. Sua utilização é pertinente para a correta utilização dos valores armazenados.

A classe *AbstractData* é definida logo abaixo, utilizando-se a linguagem de especificação CSP-OZ:



Com relação aos clientes ou objetos que efetuam acesso ao *DataBus* (figura 3.1) pode-se descrever as seguintes classes visualizadas no diagrama de classes.

- *Abstract Client*

  Na versão de leitura, o objeto cliente faz a leitura do novo dado de interesse através da classe *Concrete Client*. Na versão de escrita, o objeto *Listener* efetua a escrita no *Data Bus*, que por sua vez notifica os clientes caso exista um novo dado.

- *Concrete Client*

  É uma subclasse do *Abstract Client* que utiliza o objeto de dados que recebe de um ou mais *Abstract Subjects* através do *Data Bus*.

- *Abstract Subject*

  O *Abstract Subject* define o dado conectado ao barramento de dados. Tipicamente existem diferentes *Abstract Subject*, e cada um deles pode definir múltiplos objetos de dados. Cada objeto de dados diferencia-se dos outros

objetos através de seu *DataID* ou seu *DataName*. Normalmente, o objeto de dados com um *DataID* específico é somente definido por um único *Abstract Subject*.

- *Concrete Subject*

  O *Concrete Subject* é uma subclasse de *Abstract Subject*, responsável pela definição da classe *Abstract Subject*.

- *Data Bus*

  O *Data Bus* define um canal centralizado de objetos contendo dados a serem compartilhados. O *Data Bus* não reconhece as subclasses de dados conectadas a ela nem a sua quantidade de objetos armazenados. Na versão de escrita, o *Data Bus* possui uma lista de Notificação, uma para cada diferente *DataID* ou *DataName*. Basicamente, a lista de notificação serve para que caso ocorra uma alteração no valor de um determinado dado, outros clientes possam ser notificados da sua mudança e tomar as devidas providências. No estudo de caso não foi utilizado nenhuma classe de notificação, uma vez que o sistema não apresentou a necessidade do mesmo.

  O *Data Bus* ainda define importantes métodos como:

  - *get(DataID: ID_Type): Concrete Data*

    Na versão de escrita, o método *get* busca o dado específico ao valor de cada *DataID* e retorna o resultado ao *Abstract Client*

  - *get(DataName: String): Concrete Data*

    Na versão de escrita, o método *get* busca o dado específico de cada *DataName* e retorna o resultado ao *Abstract Client*

  - *Subscribe(DataID: ID_Type; Handle: Notification Handle)*

    Na versão de escrita, o *Listener* registra ou desregistra o dado requisitado, através da chamada de método, passando o *DataID* como parâmetro.

  - *Subscribe(DataName: String; Handle: Notification Handle)*

    Este método funciona da mesma maneira que o anterior, exceto pelo uso de *DataName* ao invés do *DataID*.

  - *Unsubscribe(DataID: ID_Type; Handle: Notification Handle)*

    Este método permite ao *Listener* desregistrar um dado específico. O parâmetro *DataID* especifica qual o dado a ser desregistrado e o *Handle* o cliente.

  - *Unsubscribe(DataName: String; Handle: Notification Handle)*

    Este método funciona da mesma maneira que o anterior, exceto pelo uso de *DataName* ao invés do *DataID*.

- *update(d: Concrete Data)*

  Este método atualiza o objeto de dado existente com o mesmo *DataID* ou *DataName* caso ele exista. Caso não exista, ele é inserido na lista de dados disponíveis do *DataBus.*

· *Listener*

O objeto *Listener* monitora o *Data Bus* através de um *pooling* periódico para um determinado dado com um *DataID* ou *DataName* específico. O *Listener* aparece apenas na versão de escrita do padrão.

· *Notification List*

Na versão de escrita do padrão, o *Data Bus* deve manter uma lista de registradores de cada objeto de dado com um *DataID* ou *DataName* específico. Cada *Notification List* gerencia uma lista de registradores, e cada registrador possui uma única *Notification Handle* para identificar como o dado pode ser escrito de maneira adequada.

· *Notification Handle*

Este objeto implementa métodos para que o *Data Bus* possa enviar dados para a classe *Abstract Client* caso seja necessário. A classe *Notification Handle* deve prover maneira de abstrair a complexidade existente na sua comunicação (por exemplo, diferentes protocolos de rede).

Abaixo, os descritivos da classe responsável pelo repositório de dados, no qual os dados lidos pelos sensores são inseridos e lidos periodicamente.

DataBus

$\backslash (INIT)$

$$dataQueue : ID_TYPE \rightarrow AbstractData$$
$$myDataBusInstance : DataBus$$

INIT

$$dataQueue = \varnothing$$
$$myDataBusInstance = NULL$$

update

$$\Delta dataQueue$$
$$data? : AbstractData$$

$$data?.DataID \in \mathrm{dom}\ dataQueue$$
$$dataQueue' = dataQueue \oplus \{data?.DataID \mapsto data?\}$$

subscribe

$$\Delta dataQueue$$
$$data? : AbstractData$$

$$data?.DataID \notin \mathrm{dom}\ dataQueue$$
$$dataQueue' = dataQueue \cup \{data?.DataID \mapsto data?\}$$

unsubscribe

$$\Delta dataQueue$$
$$data? : AbstractData$$

$$data?.DataID \in \mathrm{dom}\ dataQueue$$
$$dataQueue' = dataQueue \setminus \{data?.DataID \mapsto data?\}$$

get

$$\Xi dataQueue$$
$$dataID? : ID_TYPE$$
$$data! : AbstractData$$

$$dataID \in \mathrm{dom}\ dataQueue$$
$$data! = dataQueue\ dataID$$

get

$$\Xi dataQueue$$
$$Name? : STRING$$
$$data! : AbstractData$$

$$\exists\ i : \mathrm{dom}\ dataQueue \wedge dataQueue(i).Name = Name? \bullet data! = dataQueue(i)$$
$$\vee$$
$$\nexists\ i : \mathrm{dom}\ dataQueue \wedge dataQueue(i).Name = Name? \bullet data! = \varnothing$$

GetDataBusInstance

$$\Delta myDataBusInstance$$
$$instance! : DataBus$$

$$myDataBusInstance = NULL \bullet myDataBusInstance' = newDataBus$$
$$instance! = myDataBusInstance$$

A classe *DataBus*, especificada acima implementa o padrão de projeto *Singleton*, utilizado quando é necessário existir apenas uma referência do objeto em todo o sistema. Para se definir uma única referência da classe *DataBus*, uma única instância do objeto *DataBus* é criada, no caso, com o nome de *myDataBusInstance*, a função *GetDataBusInstance* é definida para se acessar o objeto *myDataBusInstance*, além do fato da função *INIT* não estar disponibilizada para ao meio externo.

A função *GetDataBusInstance* a priori não necessita de uma instância de um objeto do tipo *DataBus* para a sua utilização, a exemplo de algumas linguagens orientadas a objeto como Java e C++ que permitem a criação de métodos estáticos. A função *GetDataBusInstance* basicamente verifica se a variável *myDataBusInstance* já foi instanciada, caso a mesma não tenha sido, a função *GetDataBusInstance* a instancia e retorna a referência do objeto como variável de retorno. A utilização de uma semântica para a utilização de referências em Object-Z foi baseada no trabalho de Clemens Fischer (FISCHER, 2000), no qual o operador New é utilizado para se instanciar objetos.

A sua implementação basicamente é feita através da instanciação de sua referência interna, sendo que a criação direta de seu objeto não é possível, uma vez que o método *INIT*, responsável pela sua criação, não é acessível ao meio externo ($\upharpoonright (INIT)$). Qualquer objeto externo que queira utilizar a classe *DataBus* deve acessá-la através do método *GetDataBusInstance*. Caso a instância interna não tenha sido instanciada, o método de criação *INIT* é chamado e sua referência é criada caso necessário.

Para ambientes *multithreading*, pode ocorrer uma situação em que dois processos acessam o método *myDataBusInstance* pela primeira vez ao mesmo tempo, podendo ocorrer a criação de duas referências da classe *DataBus*. Para se evitar este tipo de comportamento, poderia-se utilizar uma sincronização para o método *myDataBusInstance*, o que acarretaria um excessivo *overhead* na sua utilização. Uma outra solução menos penosa, seria a chamada do método *myDataBusInstance* na iniciação do sistema, na qual poderíamos garantir que apenas um processo estaria fazendo acesso à classe.

Portanto, para a análise da criação de um repositório centralizado de estados compartilhado por todo sistema, caso existam duas instâncias para a classe *DataBus*, ambas referem-se a um único objeto.

$$
\begin{array}{|l}
dataBuses : \mathbb{P}\, DataBus \\
\hline
\forall\, db_1, db_2 : DataBus \bullet db_1 = db_2 \Leftrightarrow \\
\qquad \mu\, db : DataBus \mid db \in dataBuses
\end{array}
$$

Ou seja, o conjunto *dataBuses* possui apenas um elemento.

3.2.3 Escalonamento de Processos

O escalonamento de processos permite o compartilhamento de recursos de hardware e software entre os processos que o compõem. A utilização de prioridades para escalonamento de processos é uma abordagem clássica para sistemas de tempo real (JOSEPH, 2001).

Basicamente, a definição de prioridades de um processo é efetuada dinamicamente ou estaticamente. Prioridades dinâmicas requerem a utilização de um processo escalonador no qual as prioridades são definidas em tempo de execução de modo que os processos terminem dentro dos requisitos de tempo. Uma das suas vantagens em se utilizar este tipo de escalonamento é uma melhor utilização de recursos, porém a sua análise temporal e previsibilidade são mais difíceis do que sistemas com prioridades estáticas.

Em escalonadores estáticos, a ordem de execução de processos, bem como as suas prioridades, são definidas na fase de concepção do projeto. A seqüência de processos é armazenada em uma tabela de processos e executada conforme a definição de seus ciclos de disparo.

Escalonadores estáticos podem também ser utilizados em sistemas acionados por eventos, nos quais a seqüência de execução do escalonador estático pode ser adaptada aos eventos de entrada e a seus dados ou combinar todas as partes do processo ativadas por um evento e relacioná-lo a uma linha de execução.

Para o estudo de caso, foram adotadas a utilização e análise de escalonadores de prioridade estática, devido à sua simplicidade, estabilidade, otimização e a adequação na utilização de sistemas de tempo real.

3.2.4 Padrões de Concorrência - Padrão de prioridade estática

O padrão de prioridades estáticas é a abordagem mais comum e utilizada para escalonar sistemas de tempo real. Ela possui as vantagens de ser simples e escalonar de maneira justa uma grande quantidade de processos, além da sua simplicidade em efetuar a sua análise temporal e sua escalabilidade, utilizando métodos de análises padrões e largamente estudados no desenvolvimento de sistemas de tempo real.

Dois conceitos importantes no desenvolvimento de sistemas de tempo real são urgência e criticidade. *Urgência* se refere à proximidade de um *deadline*, enquanto que *Criticidade* refere-se a quão importante um processo deve-se enquadrar a um *deadline* para a funcionalidade do sistema. Apesar desses dois conceitos serem diferentes, normalmente os sistemas operacionais tratam esses dois problemas como um único conceito, ou seja, a prioridade. O sistema operacional utiliza a prioridade

de um processo para determinar qual processo deve ter preferência quando mais de um processo está apto a ser executado, ou seja, o sistema operacional sempre executa o processo de maior prioridade sobre todos os outros processos aptos a executar.

A abordagem mais comum para se definir prioridades em sistemas de tempo real é definir a prioridade dos processos em tempo de *design*. Esta prioridade é chamada de estática, pois ela é definida na sua concepção e não é alterada durante a execução do sistema. Esta abordagem possui as vantagens de ser simples, estável e, caso as políticas de prioridades sejam bem definidas, bastante otimizadas.

Em relação à estabilidade de um sistema, entende-se que em um sistema em tempo de execução, é possível prevermos qual processo não alcançará o *deadline* (ou seja, os processos de menor prioridade).

Em relação à otimização, um processo pode ser escalonado utilizando-se outras abordagens de modo a efetuar a melhor utilização dos recursos disponíveis, apesar da definição de prioridade de processos em escalonadores estáticos de um sistema ser um problema de complexidade NP.

O termo "Tempo Real" refere-se à previsibilidade de tempo de execução de um processo para o comportamento correto de um sistema. Esta limitação pode ser devida a uma quantidade de problemas do mundo real como, por exemplo, a quantidade de tempo que um sistema crítico pode tolerar uma falha antes da ocorrência de um incidente devido às características de estabilidade de um controlador PID. Um dos modos mais comuns de se modelar tempo para estes sistemas é definindo um intervalo de tempo, começando com a ocorrência de um evento ou o início de execução de um processo e o fim de execução deste processo, chamado de *deadline*. Sistemas podem ser analisados através da sua escalabilidade, dado a sua periodicidade, o pior tempo de execução, e o *deadline* de todos os processos, dado o conhecimento de suas prioridades.

Através do diagrama de classes da figura 3.2, pode-se destacar as seguintes classes:

- *Scheduler*

  Classe responsável pelo escalonamento do processo

- *Abstract Thread*

  Classe abstrata responsável pelo processo. Possui uma prioridade pré-definida, associada à classe *Scheduler* que por sua vez é responsável pelo escalonamento de acordo com a sua freqüência. Muitos sistemas operacionais possuem embutida a definição de prioridades fazendo com que o sistema não

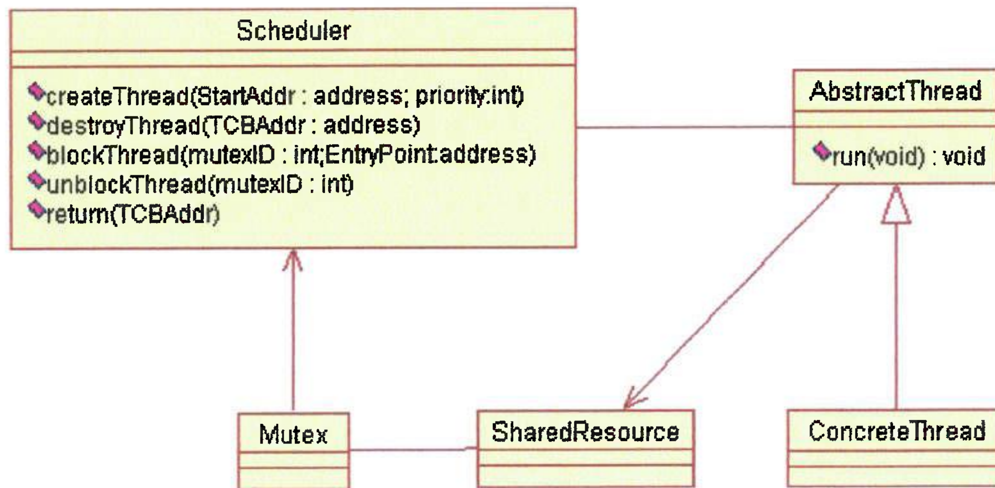


Figura 3.2: Diagrama de classes de escalonamento de processos.

seja encarregado de efetuar a alternância de processos de acordo com a sua prioridade.

- *Concrete Thread*

  Subclasse da *Abstract Thread*, responsável pela definição de cada um dos processos reais que compõem o sistema.

- *Mutex*

  Classe responsável pela exclusão mútua de um determinado recurso, permitindo que apenas um processo acesse a classe *Shared Resource*. Um processo ao efetuar uma operação em um determinado recurso compartilhado verifica se a classe *Mutex* está bloqueada ou não, caso a mesma esteja desbloqueada, a classe *Concrete Thread* a bloqueia e somente a desbloqueia ao término da operação efetuada em cima do recurso compartilhado. Qualquer processo que tente acessar um recurso compartilhado bloqueado fica em estado de espera até que o recurso se torne desbloqueado.

- *Shared Resource*

  Classe responsável pela definição do recurso compartilhado. Normalmente um recurso compartilhado possui sistemas de semáforo evitando a corrupção de dados causada por acesso simultâneo ao mesmo.

Inicialmente foram definidos os tipos de prioridades e os índices relacionados ao mesmo, dependendo de cada sistema operacional utilizado, a quantidade de possíveis prioridades é limitada. Também é definida a freqüência em que cada um dos processos é chamado pelo escalonador de processos.

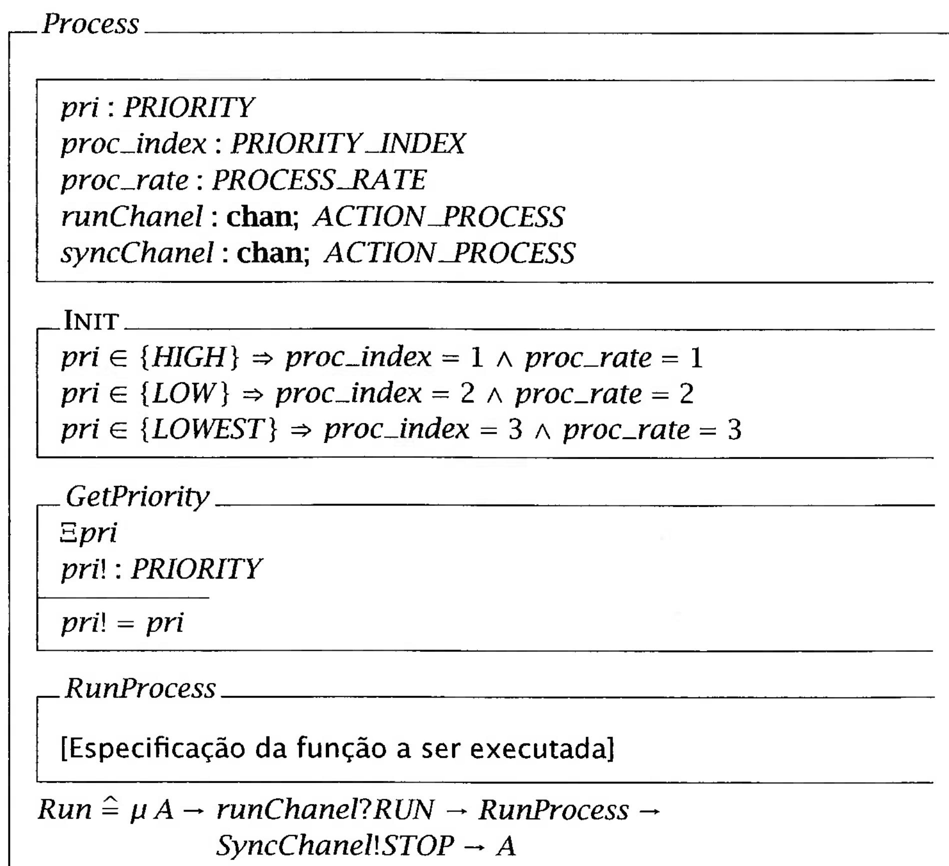
$PRIORITY ::= HIGH \mid LOW \mid LOWEST \mid NONE$
$PRIORITY_INDEX : \mathbb{N} \bullet PRIORITY_INDEX \in 1\,..\,99$
$PROCESS_RATE : \mathbb{N} \bullet PROCESS_RATE \in 1\,..\,99$

A superclasse responsável por cada um dos processos é descrita formalmente abaixo, pela classe *Process*, sendo que cada instância do objeto possui uma prioridade, um índice de prioridade, e a sua freqüência de chamadas efetuada pelo escalonador. Pode-se perceber que quanto maior a prioridade do processo, maior é a taxa em que o escalonador irá chamá-lo.

Em relação à classe *Process*, pode-se perceber a existência de dois canais (FISCHER, 1997) e (MAHONY; DONG, 2000), (*runChanel* e *syncChanel*), utilizadas pela classe escalonador para efetuar a sincronização e as chamadas dos processos, sendo utilizada a definição baseada em Fischer (1997), em que cada canal possui um tipo relacionado.

$STATE_PROCESS ::= WAITING \mid RUNNING$
$ACTION_PROCESS ::= RUN \mid STOP \mid GET_OVERFLOW_COUNTER$

A comunicação nos canais é efetuada através da utilização do tipo ACTION_PROCESS.



Process

$pri : PRIORITY$
$proc_index : PRIORITY_INDEX$
$proc_rate : PROCESS_RATE$
$runChanel : \mathbf{chan};\ ACTION_PROCESS$
$syncChanel : \mathbf{chan};\ ACTION_PROCESS$

INIT

$pri \in \{HIGH\} \Rightarrow proc_index = 1 \wedge proc_rate = 1$
$pri \in \{LOW\} \Rightarrow proc_index = 2 \wedge proc_rate = 2$
$pri \in \{LOWEST\} \Rightarrow proc_index = 3 \wedge proc_rate = 3$

GetPriority

Ξpri
$pri! : PRIORITY$

$pri! = pri$

RunProcess

[Especificação da função a ser executada]

$Run \mathrel{\widehat{=}} \mu A \rightarrow runChanel?RUN \rightarrow RunProcess \rightarrow SyncChanel!STOP \rightarrow A$

Cada um dos processos definidos sincroniza através de um processo sincronizador responsável pela definição do estado do processo em si. O processo sincro-

nizador pode ser formalizado pela classe *SyncProcess*, que por sua vez é disparado pela classe *Scheduler*.

A classe *SyncProcess*, além dos canais syncChannel e runChannel comuns a cada processo, ainda possui um canal de segurança no qual é informada a quantidade de *overflow* que o processo executou. Entende-se por *overflow* a situação em que o processo foi escalonado pela classe *Scheduler* e o mesmo ainda não terminou o seu ciclo de execução, ou seja, o requisito de tempo pré-definido não foi alcançado.

Ainda em relação à classe *SyncProcess*, pode-se observar a existência da variável de estado *overFlowCounter*, responsável por guardar a quantidade de *overflow*, alterada através das operações *AddOverflowCounter* e lidas a partir de *GetOverflowCounter*. A variável de estado responsável pelo estado atual do processo é a variável *myState* do tipo *STATE_PROCESS*. A alteração desta variável de estado é efetuada pelo canal *syncChannel*, escritas pelas classes *Scheduler* (para disparar o início de um novo ciclo) e pela classe *Process* (através da notificação do fim do ciclo).

Em relação ao relacionamento existente entre os canais de sincronização de processos e os canais de notificação de segurança, a prioridade do canal de sincronização de processos possui uma prioridade maior do que o canal de notificação de *overflow*, isso é especificado de acordo com a notação $\mathcal{CSPP}$ (LAWRENCE, 2003b, 2002, 2004, 2003a), a respeito de prioridades sobre escolhas externas $\overleftarrow{\Box}$.

A semântica relacionada a $\mathcal{CSPP}$ difere um pouco em relação à semântica CSP através da introdução da semântica *Acceptances* (LAWRENCE, 2002), na qual é analisado o comportamento do processo através da aceitação de um subconjunto de eventos oferecido ao processo. No caso acima, caso sejam oferecidos os eventos *syncChannel.Run* e *securityChannel.RUN*, o sistema aceitaria o evento *syncChannel.Run*.

$$\langle\rangle : \{syncChannel.Run, securityChannel.RUN\} \leadsto \{syncChannel.RUN\}$$

SyncProcess

$syncChannel : \mathbf{chan};\ ACTION_PROCESS$
$runChannel : \mathbf{chan};\ ACTION_PROCESS$
$securityChannel : \mathbf{chan};\ ACTION_PROCESS$
$reportchannel : \mathbf{chan};\ INTEGER$
$myState : STATE_PROCESS$
$overFlowCounter : INTEGER$

INIT

$overFlowCounter = 0$
$myState = WAIT$

AddOverflowCounter

$\Delta overFlowCounter$

$overFlowCounter' = overFlowCounter + 1$

GetOverflowCounter

$\Xi overFlowCounter$
$overFlowCounter!$

$overFlowCounter! = overFlowCounter$

StopProcess

$\Delta myState$

$myState' = WAITING$

StartProcess

$\Delta myState$

$myState = WAITING \bullet$
$\quad myState' = RUNNING \wedge runchannel!(RUN)$
$myState = RUNNING \bullet AddOverflowCounter$

$$
\begin{array}{l}
Run \mathrel{\widehat=} \mu A \rightarrow (syncchannel?(action) \rightarrow \\
\quad (\\
\qquad (StopProcess \rightarrow A) \\
\qquad \Box \\
\qquad (StartProcess \rightarrow \\
\qquad\quad ((runchannel!(RUN) \rightarrow A) \\
\qquad\quad \sqcap \\
\qquad\quad (AddOverflowCounter \rightarrow A))) \\
\quad) \\
) \\
\overleftarrow{\Box} \\
(securityChannel?(action) \rightarrow \\
\quad GetOverflowCounter \rightarrow \\
\quad reportchannel!overFlowCounter \rightarrow A \\
)
\end{array}
$$

A classe *Scheduler* por sua vez, possui as seguintes características:

- Define a tabela contendo todos os processos cadastrados a ela, bem como a

sua taxa de execução e prioridade.

- Para se utilizar o conceito de tempo em relação à variável *CYCLE_TIME*, foi utilizada a notação especificada de TCOZ em (MAHONY; DONG, 2002), na qual ao término de cada iteração, o processo de escalonamento permanece parado até que o novo ciclo seja reiniciado.

Para se implementar uma lista de objetos finita, utilizaremos a seguinte definição para a classe genérica:

$$[X]$$
$$Array : \mathbb{P}(\mathbb{N} \rightarrow X)$$
$$Array = (1..n) \rightarrow X$$

Sendo que a classe *Scheduler* possui uma *Array*[*syncChannel*], ou seja, um canal de sincronização para cada processo.

Scheduler

$$CYCLE_TIME : \mathbb{N}$$
$$CYCLE_TIME = 50$$

$$countCycle : \mathbb{N}$$
$$syncChannelArray : Array[\mathbf{chan};\ ACTION_PROCESS]$$
$$rateOfPriority : PRIORITY \rightarrow PROCESS_RATE$$
$$indexOfPriority : PRIORITY \rightarrow PROCESS_INDEX$$

Scheduler do sistema

INIT

$$countCycle = 0$$

RunScheduler

$$\Delta(countCycle)$$
$$\exists\, index : \mathbb{N} \land (countCycle \bmod index) = 0 \land index \in \operatorname{dom} syncChannelArray \bullet syncChannelArray(index)!RUN$$
$$countCycle' = countCycle + 1$$

$$Run \mathrel{\widehat=} \mu R \rightarrow RunScheduler \rightarrow \text{WAIT}\ CYCLE_TIME \rightarrow R$$

Levando em consideração que todo os objetos ativos (MAHONY; DONG, 1998) relacionados aos processos são derivados da super classe *Process*, e sob o ponto de vista da classificação de subtipos elaborada por Wehrheim (WEHRHEIM, 2000; FISCHER; WEHRHEIM, 2000; WEHRHEIM, 2001), é desejável que a classe *Process* seja classificada como um tipo ótimo, em que a herança pode introduzir novos métodos, porém, de uma maneira "amigável", de modo a não introduzir novos estados na classe pai, ou seja, a classificação de uma herança ótima estaria relacionada mais à idéia de uma

implementação, no caso o método *RunProcess*, do que uma extensão em si, fazendo com que os novos métodos sejam responsáveis pelo comportamento da classe pai, não introduzindo falhas, divergências e novos estados.

De acordo com Wehrheim (2001), a semântica em relação a classes otimísticas seriam:

Seja $F \subseteq \Sigma^* \times 2^{\Sigma}$ o conjunto de falhas e $N \subseteq \Sigma$ o conjunto de eventos e A, C especificações CSP, sendo A relacionado à super classe e C a classe derivada e seja $N = \alpha(C) \setminus \alpha(A)$ o conjunto de novos métodos.

- $F \setminus_c N = \{(\sigma, X) \mid \exists(\sigma', Y) \in F, \sigma = \sigma' \downarrow (\Sigma \setminus N), X \subseteq Y \cup N\}$
- *C é uma classe ótima de A (escrita $A \sqsubseteq^N_{ost} C$) se $failures(C) \setminus_c N \subseteq failures(A)$*

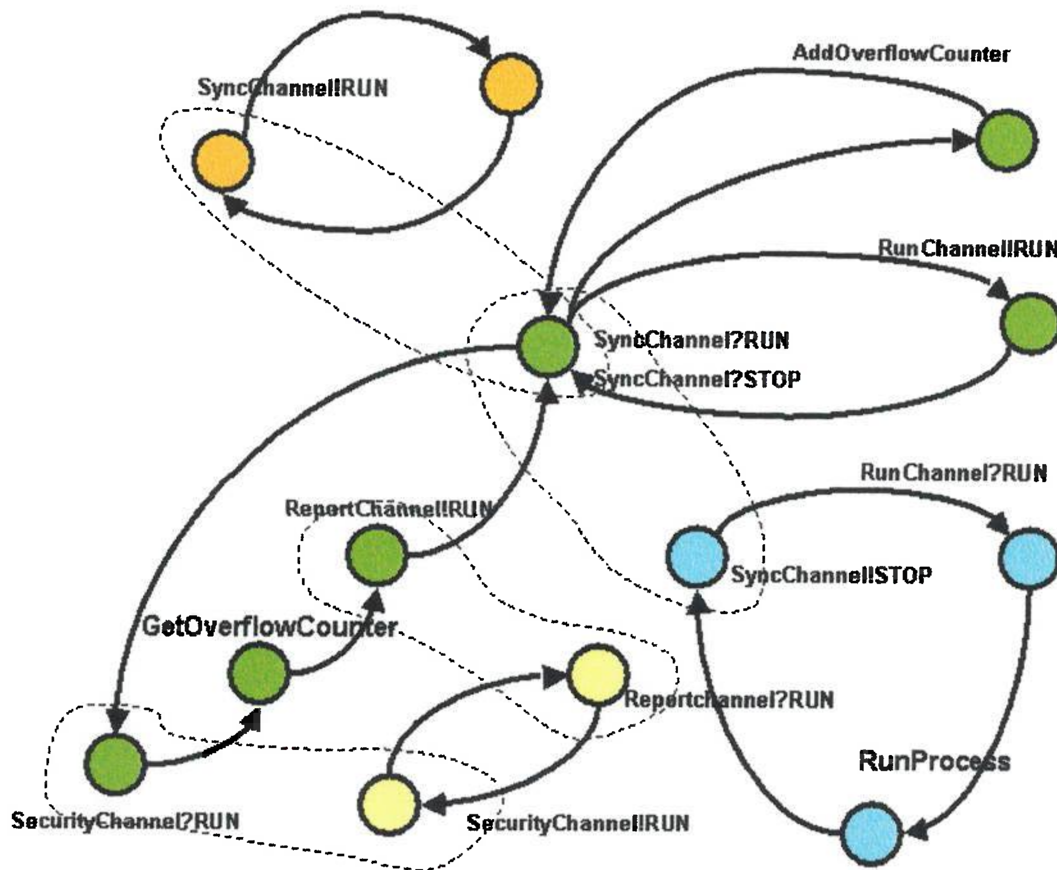


Figura 3.3: Diagrama entre os processos.

Na figura 3.3, é mostrado um diagrama para cada uma das classes envolvendo a equação 3.1, sendo basicamente mostrados os processos de escalonamento, representados pelas classes *Scheduler* em laranja, o processo de sincronizaçao representado pela classe *syncProcess* em verde, e o processo derivado da classe *Process* responsável pelo processo em si. O diagrama em amarelo representa o monitoramento do sistema, feito pela classe *WathDog*. A linha pontilhada representa o ponto onde é efetuada a sincronização entre os canais.

O ponto crítico do sistema, portanto, fica ao encargo da implementação efetuada através de herança do método *RunProcess*, uma vez que ela pode levar a divergências ou *deadlocks* devido a uma má implementação.

Pode-se perceber também que os requisitos de tempo do sistema estão intimamente relacionadas com o tempo em que se leva para terminar cada um dos ciclos definido pelo método *RunProcess*.

Uma análise inicial do escalonamento envolveria a análise de *failures* e *divergencies* da super classe *Process* em relação às classes *Scheduler* e *SyncProcess*.

Utilizando-se a notação CSP, pode-se formalizar como:

$$sistema = scheduler \,|[\, syncChanel \,]|\, ((syncProcess \,|[\, runChanel \,]|\, Process) \quad (3.1)$$

Processos cíclicos compartilhando canais de sincronização, em que em cada ciclo ocorre a comunicação de todos os seus canais de comunicação são livres de *deadlock* (MARTIN, 1996), sendo que cada evento não requer a participação de mais de dois processos (ROSCOE, 1998). Pode-se perceber que as classes *Scheduler* e *Process* apresentam esse comportamento.

Em relação à classe *SyncProcess*, apesar da comunicação entre outras classes ser efetuada pelo canal *syncChanel*, para iniciar e notificar a parada do processo, pode-se perceber que ela poderia ser decomposta em 2 canais independentes.

Tomando como analogia a definição de funções recursivas para Object-Z e aplicando ao seu menor valor do reticulado (SMITH, 2000b) como [false] para a equação recursiva definida na classe *SyncProcess*, é fácil verificarmos que se trata de uma função recursiva que não pode terminar e similar à operação [false].

A natureza infinita da função permite pelo menos analisar a natureza dos *traces* correspondentes à sua execução. Para visualizarmos as possíveis transições de estado da equação 3.1, os seguintes *traces* podem ser gerados:

$$Trace^0(sistema) = \{\langle\rangle\}$$

$$Trace^1(sistema) = \{\langle syncChannel.RUN \rangle\}$$

$$Trace^2(sistema) = \{\langle syncChannel.RUN, runChannel.RUN \rangle\}$$

$$\begin{aligned} Trace^3(sistema) = \{ & \\ & \langle syncChannel.RUN, runChannel.RUN, \\ & \quad syncChannel.RUN \rangle, \\ & \langle syncChannel.RUN, runChannel.RUN, \\ & \quad RunProcess \rangle \} \end{aligned}$$

$$
\begin{aligned}
Trace^4(sistema) = \{ \\
&\langle syncChannel.RUN, runChannel.RUN, syncChannel.RUN, \\
&\quad addOverflowCounter\rangle, \\
&\langle syncChannel.RUN, runChannel.RUN, \\
&\quad syncChannel.RUN, RunProcess\rangle, \\
&\langle syncChannel.RUN, runChannel.RUN, \\
&\quad RunProcess, syncChannel.RUN\rangle, \\
&\langle syncChannel.RUN, runChannel.RUN, \\
&\quad RunProcess, syncChannel.STOP\rangle\}
\end{aligned}
$$

$$
\begin{aligned}
Trace^5(sistema) = \{ \\
&\langle syncChannel.RUN, runChannel.RUN, syncChannel.RUN, \\
&\quad addOverflowCounter, syncChannel.RUN\rangle, \\
&\langle syncChannel.RUN, runChannel.RUN, syncChannel.RUN, \\
&\quad addOverflowCounter, RunProcess\rangle, \\
&\langle syncChannel.RUN, runChannel.RUN, syncChannel.RUN, \\
&\quad RunProcess, addOverflowCounter\rangle, \\
&\langle syncChannel.RUN, runChannel.RUN, RunProcess, \\
&\quad syncChannel.RUN, addOverflowCounter\rangle, \\
&\langle syncChannel.RUN, runChannel.RUN, RunProcess, \\
&\quad syncChannel.STOP, syncChannel.RUN\rangle\}
\end{aligned}
$$

Ou seja, é possível encontrarmos uma equação genérica para o comportamento do sistema, de modo a encontrarmos uma solução para a função recursiva demonstrada abaixo:

$$F^0(x) = syncChannel.RUN \rightarrow F^1(x)$$

$$F^1(x) = runChannel.RUN \rightarrow F^2(x)$$

$$F^2(x) = (syncChannel.RUN \rightarrow F^3(x)) \sqcap (RunProcess \rightarrow F^4(x))$$

$$F^3(x) = (addOverflowCounter \rightarrow F^2(x)) \sqcap (RunProcess \rightarrow F^5(x))$$

$$F^4(x) = (syncChannel.RUN \rightarrow F^5(x)) \square (syncChannel.STOP \rightarrow F^0(x))$$

$$F^5(x) = (addOverflowCounter \rightarrow F^4(x))$$

Os possíveis estados estão mostrados na figura 3.4. Cada um dos possíveis estados alcançáveis pelo sistema, são descritos, de acordo com a semântica de verificação de sistemas através de *failures*, sendo *failures* definido pelo par (tr, X), onde tr é o possível conjunto de *traces* e X o conjunto de eventos recusados pelo sistema de modo a ocorrer um *deadlock*.

$$failures = \{F^0(x), \langle syncChannel.RUN\rangle \notin X\}$$

$$failures = \{F^1(x), \langle runChannel.RUN\rangle \notin X\}$$

$$failures = \{F^2(x), \langle syncChannel.RUN, RunProcess\rangle \notin X\}$$

$$failures = \{F^3(x), \langle addOverflowCounter, RunProcess\rangle \notin X\}$$

$$failures = \{F^4(x), \langle syncChannel.RUN, syncChannel.STOP\rangle \notin X\}$$

$$failures = \{F^5(x), \langle addOverflowCounter\rangle \notin X\}$$

Uma vez que para cada estado alcançável pelo sistema, existe pelo menos um evento estável aceitável pelo sistema, pode-se concluir que o sistema é livre de *deadlock*. Como para cada ocorrência de evento no sistema o mesmo se encontra em um estado estável, a ocorrência de divergência também é nula.

Toda a classe derivada da classe *Process* deve implementar o método *RunProcess* de maneira otimística (WEHRHEIM, 2001), ou seja, de modo a não introduzir novos estados no sistema analisado acima, porém, é bom observarmos que a implementação da classe derivada pode introduzir *deadlocks* locais ou divergências, na qual pode ser observada pelo aumento do contador de *overflows* de maneira infinita.

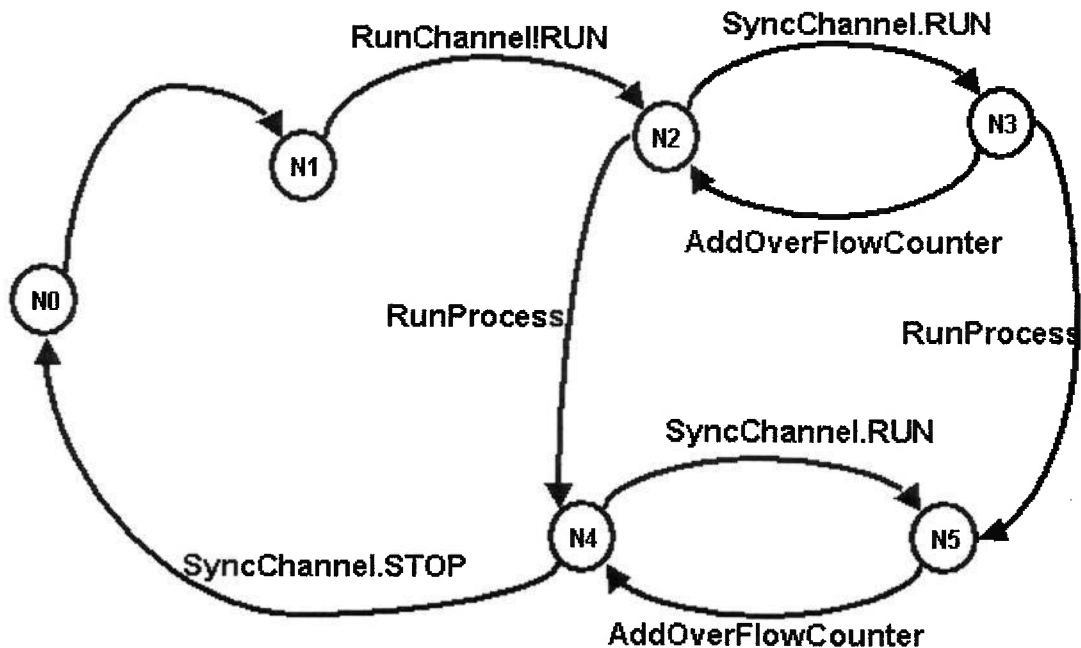
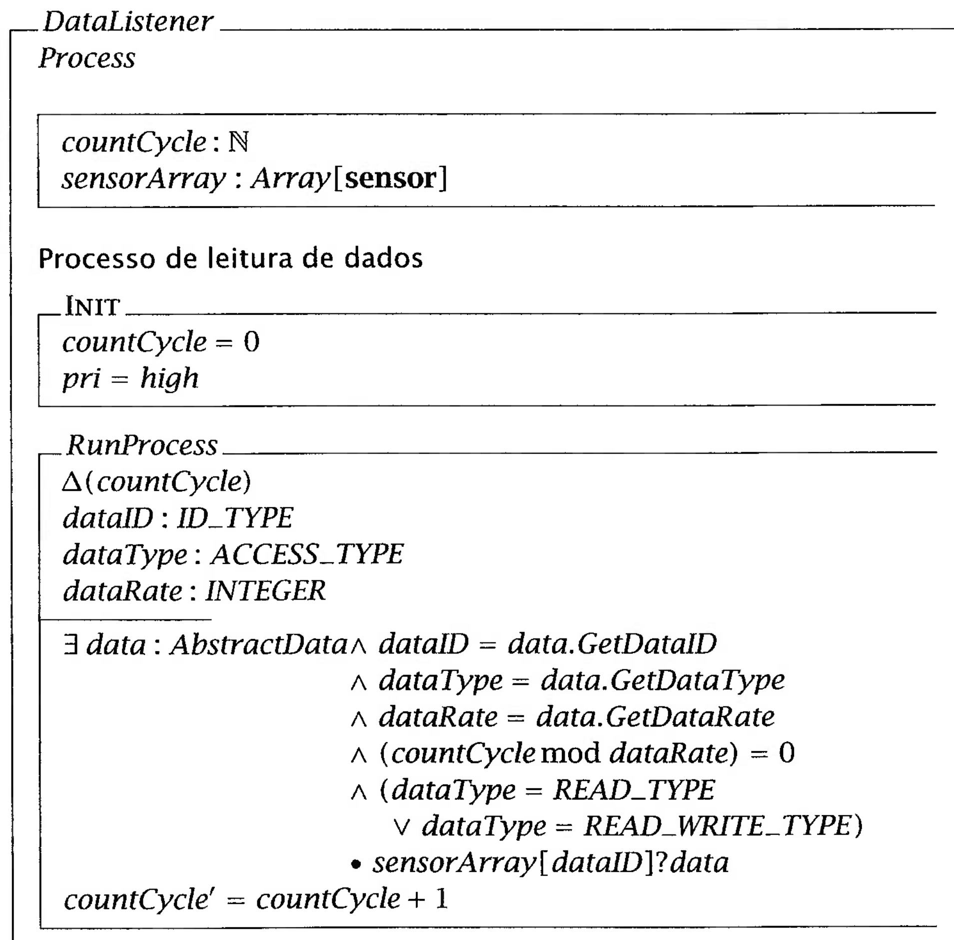
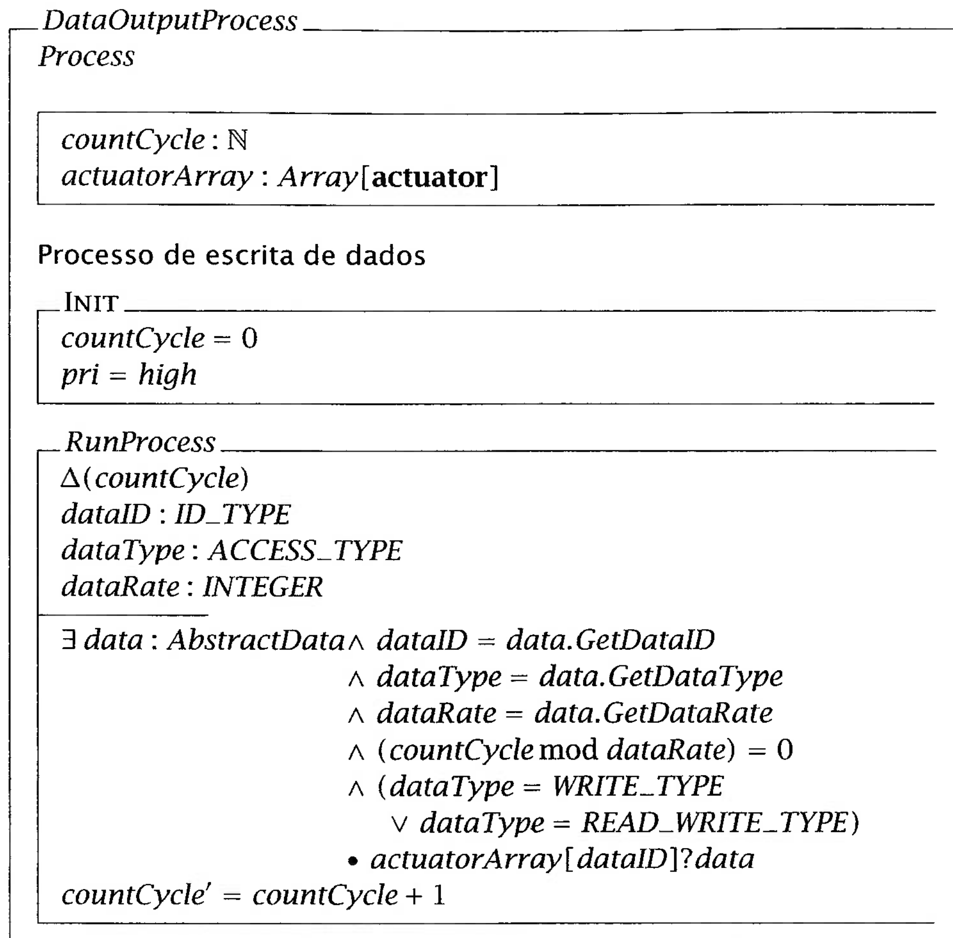


Figura 3.4: Diagrama de estados.

3.2.5 Processos de leitura e escrita de dados

Dentre os processos que compõem um sistema de tempo real, pode-se destacar os processos relacionados à leitura e escrita de dados na classe *DataBus*. No modelo adotado por este trabalho, podemos destacar os processos *DataListener*, responsável pela leitura de dados e o processo *DataOutputProcess*, responsável pela escrita de dados, lidos a partir da classe *DataBus*. A comunicação efetuada por cada um dos possíveis dados cadastrados na classe *DataBus* são abstraídos através da utilização do canal **sensor**, para a leitura de dados e **actuator** (MAHONY; DONG, 1998, 2000, 2002), para escrita de dados.





Pode-se visualizar através dos esquemas acima, que tanto os processos de leitura e de escrita são classes derivadas da classe *Process*, cujas freqüências são as mais altas do sistema, sendo que a leitura ou escrita dos seus respectivos valores são definidas pela definição da variável *DataRate* da classe *AbstractData*, ou seja , a cada ciclo disparado pela classe *Scheduler*, os processos acima verificam os dados a serem atualizados conforme a sua definição.

3.2.6 Padrões de Segurança

A possibilidade de ocorrência de falhas em sistema de tempo real deve ser tratada de maneira a prevenir acidentes. Basicamente, a implementação de segurança é efetuada através de algum tipo de redundância.

As falhas podem ter a seguinte natureza:

- Sistemática: Ocorrido devido a erros de concepção e definição do sistema. Normalmente se referem ao software desenvolvido.
- Aleatória: Erro devido a quebras ou mau funcionamento de equipamentos. Normalmente relacionados a equipamentos mecânicos ou elétricos.

Normalmente, os erros sistemáticos não possuem proteção à redundância, uma vez que o software que apresenta erros possui as mesmas características do sistema redundante e, conseqüentemente, os dois sistemas a priori possuem os mesmos erros.

Na ocorrência de falhas, os seguintes procedimentos podem ser tomados pelo sistema:

- A operação que apresenta erros reenvia os procedimentos que apresentaram falhas.
- Utilização de informação redundante para corrigir o erro.
- Mudança do estado do sistema para um estado seguro. Normalmente o estado seguro de muitos sistemas é o desligamento dos mesmos.
- Alerta de emergência para o operador.
- Reinicialização do sistema.

A redundância de sistema pode ser realizada através dos seguintes padrões:

- Redundância Homogênea: Em que o mesmo sistema é duplicado ou triplicado e em caso de falhas os procedimentos de alternância são efetuados. Uma característica da redundância é que todos os sistemas apresentam as mesmas características de equipamento e software.
- Redundância Heterogênea: Equivalente à Redundância Heterogênea, com a exceção que as redundâncias possuem naturezas diferentes, sejam elas no equipamento ou até no software. Por exemplo, um sistema pode ter equipamentos elétricos e mecânicos de naturezas diferentes, hardwares de natureza diferente, sistemas operacionais diferentes e softwares implementados em diferentes linguagens. Esta redundância implica em um maior custo de implementação, normalmente utilizados em sistemas nos quais a confiabilidade é extremamente necessária.

3.2.7 Implementação de um *WatchDog*

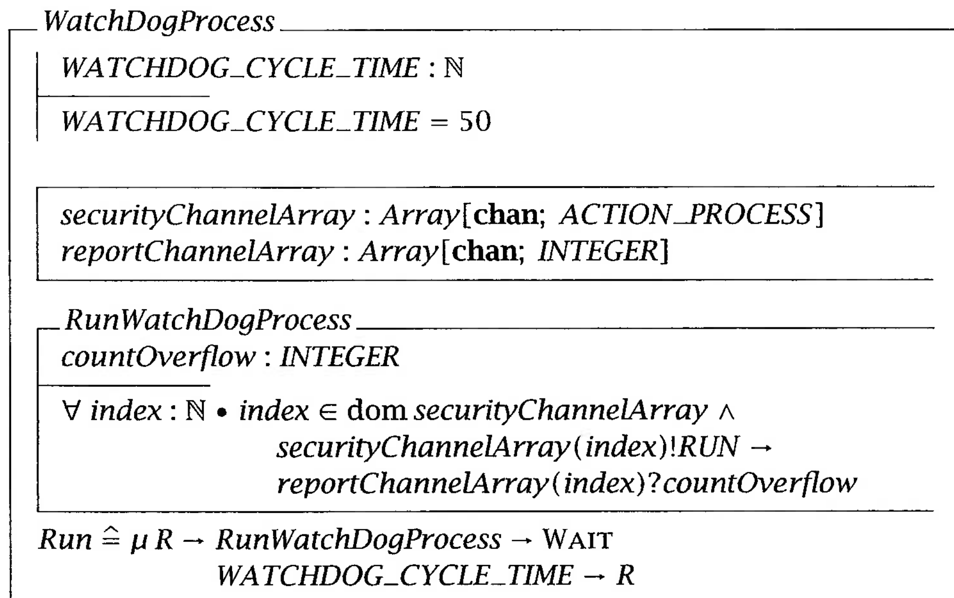
Juntamente com a utilização de redundâncias, um padrão de segurança bastante comum em sistemas de tempo real é a utilização de um sistema que verifica periodicamente o comportamento dos processos que a compõem, se os mesmos atendem as características de tempo definidas no *design* do sistema. Muitas vezes um

processo pode entrar em *deadlock* ou *livelock* devido a problemas de mau funcionamento de algum equipamento e nestes casos, um procedimento de monitoramento se faz necessário.

Esta implementação recebe o nome de *watchdog* (cão de guarda). Como o próprio nome diz, basicamente ela verifica as variações de tempo de execução dos processos de modo a certificar o bom funcionamento do sistema.

Uma parte da implementação do *watchdog* pode ser visualizada na seção 3.2.3, através do canal *securityChannel*, que consulta a quantidade de vezes em que um processo deixou de atender os requisitos de tempo. Este mesmo canal é utilizado pela classe *WatchDogProcess* para consultas periódicas dos processos. A sua utilização se torna obrigatória, uma vez que a classe *Process*, como descrita na seção 3.2.3, poderia introduzir divergências. A utilização de um *watchdog* efetuaria uma verificação periódica dos processos.

Abaixo, podemos visualizar o esquema correspondente à classe *WatchDogProcess*.



É bom frisarmos que a classe *WatchDog* não é derivada da classe *Process* e é paralela à classe *Scheduler*, sincronizando através dos canais *securityChannel*, ou seja:

$$Scheduler \mid[\ securityChannelArray\ ]\mid WatchDogProcess \tag{3.2}$$

3.2.8 Tratamento de Eventos Assíncronos

No tratamento de eventos assíncronos gerados pelo sistema, as seguintes abordagens podem ser adotadas:

- A utilização de um sistema de consultas para verificar se algum estado do sistema foi alterado. Esta solução é basicamente um tratamento síncrono de eventos, uma vez que forçamos as entradas a ocorrerem dentro de um limite de tempo pré-determinado. A escolha do tempo entre consultas deve ser definida adequadamente para que a ocorrência de eventos não seja perdida pelo sistema.
- A utilização de interrupções, em que a ocorrência de um evento gera uma rotina de interrupção de serviço no processador. Normalmente estas rotinas são pequenas, de modo a não influenciar o comportamento global do sistema, ou simplesmente um pré-tratamento dos eventos, sendo os mesmos enfileirados em uma fila pré-determinada para tratamento futuros.
- A utilização de um processo de leitura de eventos para serem enfileirados em uma fila pré-determinada para tratamentos futuros. Basicamente ela é igual ao item anterior, com a exceção que a mesma não utiliza uma interrupção para a leitura de eventos.

A utilização de um repositório de eventos, ou seja, filas de eventos com política de tratamento pré-determinada como, por exemplo, uma fila FIFO (*First In First Out*), apresenta uma flexibilidade maior em relação à utilização de um sistema de consulta, uma vez que a mesma desacopla o processamento do evento em relação à leitura de ocorrência de eventos, permitindo-se desta maneira uma análise mais simples e correta do tempo de processamento necessário para se tratar o evento. A terceira opção de tratamento de eventos assíncronos possui uma flexibilidade maior em relação à segunda opção, uma vez que a mesma não chega a entrar em detalhes de hardware, necessário para se implementar uma rotina de interrupção.

Para se implementar uma política de tratamento de eventos através da utilização de filas, é necessário que os eventos a serem enfileirados possuam a mesma prioridade. Uma outra abordagem seria a utilização de filas com prioridades. Neste tipo de filas, os eventos são explicitamente separados por equipamento, endereço e ordem de prioridade, bem como tolerância de tempo pré-determinado. Tanto a sua leitura para enfileiramento como o seu tratamento poderia ser executado por diversos processos paralelos, ou um único processo de leitura de fila, que determinaria a política de tratamento de eventos de acordo com as necessidades de alocação de recursos. Quanto maior a complexidade de tratamento de filas de um sistema, maior

é a possibilidade do sistema apresentar erros, *deadlocks*, mensagens perdidas ou condições de divergências de processos.

A especificação de um processo de tratamento de eventos assíncronos através da utilização de operadores CSP poderia ser feita através de um processo aguardando a chegada de ocorrências na sua fila por um tempo pré-determinado. A especificação, neste caso, utiliza-se de uma escolha externa, disparável após um tempo pré-determinado.

Um exemplo de sua especificação pode ser visualizado logo abaixo.

$$EVENTOS_CHANNEL_TYPE ::= BUTTON_TYPE \mid EMERGENCY_TYPE \mid \ldots$$

EventosProcess
Process

$WAIT_TIME : \mathbb{T}$

$WAIT_TIME : 50$

$$RunProcess \mathrel{\widehat=} ([v : EVENTOS_CHANNEL_TYPE] \bullet processaEvento) \;\Box\; (\textsc{Wait}\ WAIT_TIME \rightarrow \textsc{Skip})$$

A classe de tratamento de eventos acima espera a chegada de novos eventos durante *WAIT_TIME*, caso nenhuma ocorrência de evento seja disparada, o processo termina sem executar. Subseqüentes chamadas a esse processo são executadas pelo escalonador de processos.

4 Estudo de caso

Este capítulo tem por finalidade empregar os conceitos e análises dos padrões de projeto descritos no capítulo anterior em um estudo de caso, bem como a utilização de linguagens de especificação formal descritos no capítulo 2. Na seção 4.1, o estudo de caso é descrito, e na seção 4.2, os aspectos dinâmicos e estáticos do estudo de caso são descritos.

4.1 Introdução

O estudo de caso considerado é o sistema supervisório de produção de açúcar na Usina Cruz Alta do Grupo Açúcar Guarani S/A, localizada no município de Olímpia - SP, que pode ser mais bem descrito por Villani (2003). Devido à sua alta complexidade e diversidade de processos que compõem a produção industrial de açúcar, foi levada em consideração apenas uma parte do processo, no caso, o processo de carregamento de cana-de-açúcar.

4.1.1 Descrição do sistema

De acordo com a descrição realizada por Villani (2003), o processo de produção de açúcar pode ser organizado nas seguintes etapas:

- Recepção - A cana-de-açúcar chega à usina em caminhões e pode ser destinada diretamente para a produção ou pode ser armazenada em um fosso para posterior processamento. A cana enviada à produção é lavada e despejada numa esteira transportadora.
- Preparo - O preparo consiste essencialmente em passar a cana que está sendo transportada da esteira por um picador e um desfibrador. Após o desfibrador, uma segunda esteira é responsável por levar a cana até o processo de extração.
- Ao contrário de muitas usinas de cana-de-açúcar no Brasil que utilizam moendas, na Usina Cruz Alta a extração do caldo é realizada através de difusão. A difusão é o processo pelo qual a cana picada e desfibrada é mergulhada

em uma solução menos concentrada que o caldo nelas contido e cede a esta solução uma parte ou a totalidade do açúcar que constitui o excedente de concentração de seu caldo. A difusão é realizada no difusor, um equipamento formado por um transportador horizontal sobre o qual se coloca a cana a ser processada. Durante todo o seu trajeto dentro do difusor, esta camada de cana é abundantemente regada com solução, extraindo assim o caldo da cana.

- Tratamento do caldo - O caldo extraído passa por um processo de sulfitação (adição de dióxido de enxofre - SO_2), calagem (adição de cal - CaO) e aquecimento. O objetivo destes processos é esterilizar, regular o pH e ajustar a coloração do caldo.

- Clarificação - O caldo proveniente dos aquecedores é enviado aos clarificadores, que são decantadores na forma de tanques onde se faz chegar de modo regular e contínuo o caldo a ser decantado. Eles são suficientemente grandes para que a velocidade de escoamento e de circulação do caldo seja lenta o bastante para permitir a decantação.

- Evaporação - O caldo clarificado contém cerca de 85% de água. A evaporação deve eliminar a maior parte desta água. Ela é realizada em evaporadores do tipo múltiplo-efeito. O produto resultante, de alta densidade, é chamado de xarope.

- Centrifugação - A massa produzida pelo cozedor é destinada às centrífugas, onde o restante da água é resfriado.

- Secagem e empacotamento - O açúcar em cristais obtido após a centrifugação é enviado ao secador e por fim ao empacotamento, onde o açúcar produzido é armazenado em sacos.

4.1.1.1 Requisitos do sistema

Em relação aos requisitos do sistema de carregamento de cana-de-açúcar, utilizado no estudo de caso, o sistema deve:

- Decidir o destino dos caminhões que chegam à usina, ou seja, ao fosso ou à esteira de produção.

- Solicitar a transferência de cana do fosso para a produção quando necessário, o que ocorre normalmente no período noturno.

- Interromper o fornecimento de cana, desligando as mesas alimentadoras e os respectivos chuveiros em caso de alarme.

O descarregamento dos caminhões de cana é realizado por dois guindastes, fixos ao solo, um ao lado do fosso e outro ao lado da produção. Em ambos os casos, existem duas formas de descarregar a cana: por tombamento da caçamba do caminhão ou através de um sistema de correntes. Para transportar a cana do fosso para a produção utiliza-se um transportador (com garras) que não necessita de guindaste para carregar e descarregar a cana.

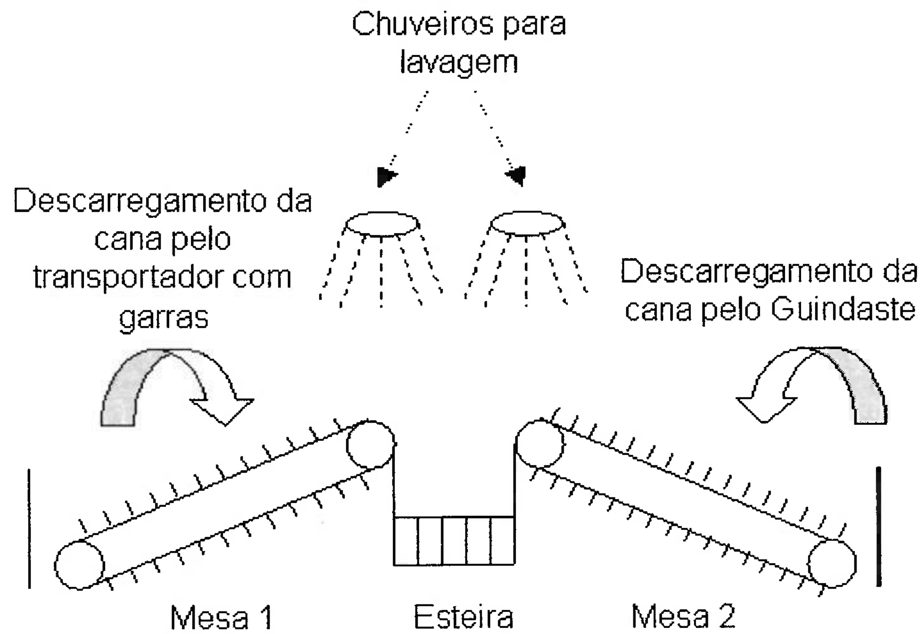


Figura 4.1: Recepção de cana.

A cana retirada do caminhão é colocada nas duas mesas de alimentação: a primeira recebe a cana proveniente do fosso e a segunda recebe cana diretamente dos caminhões que chegam à usina. Em cada uma das mesas, a cana é lavada através do acionamento de chuveiros.

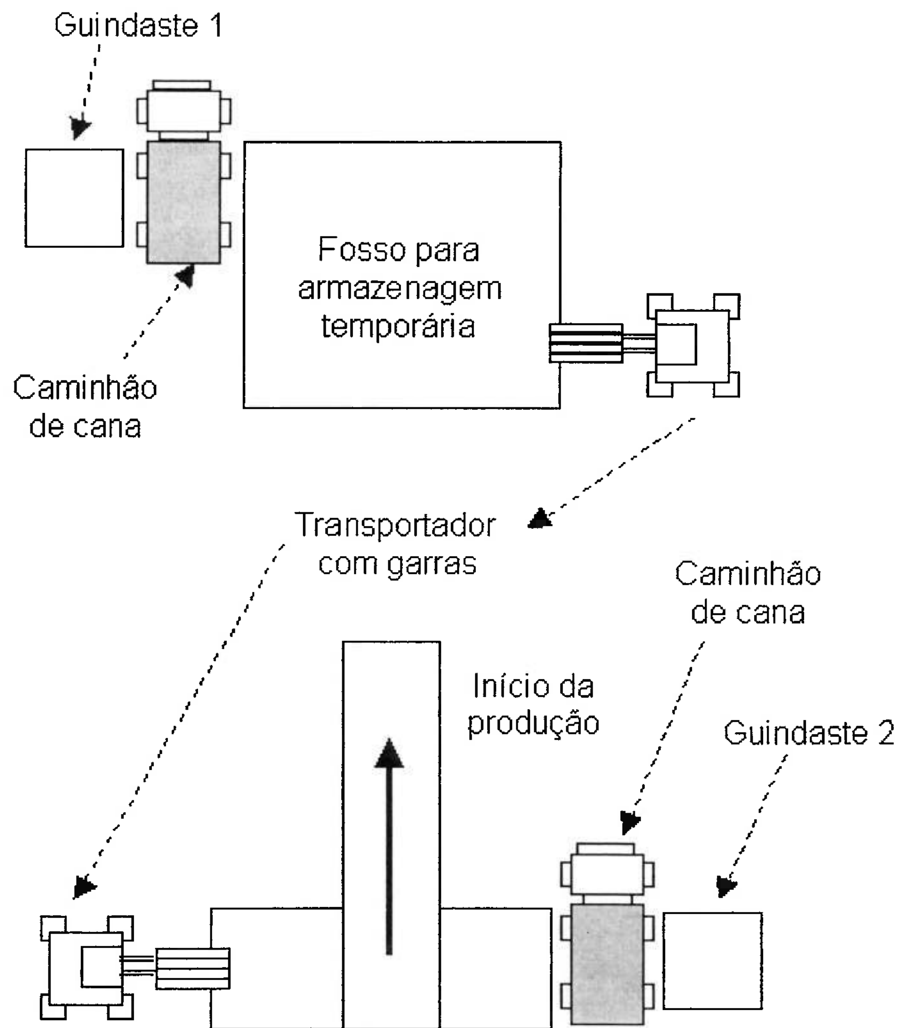


Figura 4.2: Recepção e lavagem de cana.

4.2 Processos que compõem o estudo de caso

Através da análise dos requisitos do sistema de carregamento de cana-de-açúcar, é possível definirmos os dados de monitoramento e os sistemas de atuação que a compõem.

São definidos abaixo os tipos de dados guardados na classe *DataBus*, todos eles são definidos como do tipo *ID_TYPE*. Para se efetuar o carregamento foram analisados 6 tipos de dados referentes aos sensores ou dados de entrada pertinentes ao sistema, ou seja, o dado referente à existência ou não de caminhão para se efetuar o descarregamento, o tipo de caminhão, a quantidade de feixes, o volume de feixes e o peso existente em cada uma das mesas que compõem o transporte da cana-de-açúcar para a produção e a existência de cana-de-açúcar no fosso. Os sensores responsáveis pela quantidade de feixes e pelo volume de feixes são referentes apenas a caminhões com corrente.

Em relação aos atuadores, pode-se definir os atuadores relacionados ao caminhão (caçamba), ao guindaste (gancho, cabo e rotacionador), aos atuadores dos chuveiros e os atuadores das velocidades da mesa.

ID_TYPE ::= *SENSOR_CAMINHAO_EXISTE*
| *SENSOR_TIPO_CAMINHAO*
| *SENSOR_GANCHO*
| *SENSOR_ROTACIONADOR*
| *SENSOR_CABO*
| *SENSOR_CANA_FOSSO*
| *SENSOR_NFEIXES_CAMINHAO*
| *SENSOR_PESO_MESA_1*
| *SENSOR_PESO_MESA_2*
| *SENSOR_VOLUME_FEIXES*
| *ATUADOR_GANCHO*
| *ATUADOR_ROTACIONADOR*
| *ATUADOR_CABO*
| *ATUADOR_CACAMBA*
| *ATUADOR_CHUVEIRO*1
| *ATUADOR_CHUVEIRO*2
| *ATUADOR_VELOCIDADE_MESA*1
| *ATUADOR_VELOCIDADE_MESA*2

Os tipos *UNIT_TYPE* e *INFO_TYPE* basicamente definem a unidade de valor e informações dos tipos guardadas pelas variáveis respectivamente.

INFO_TYPE ::= *SHORT* | *INT* | *LONG* | *FLOAT* | *DOUBLE* | *BOOLEAN*
UNIT_TYPE ::= *METROS* | *SEGUNDOS* | *CELSIUS* | *NONE* | M^3 | *KG*

Através da definição dos tipos acima, pode-se definir os seguintes tipos utilizados no *DataBus* enumerados nas tabelas 4.1 e 4.2. As classes de dados herdam

todos os operadores de *AbstractData*, sendo definidas também, as taxas de atualização de cada um dos dados.

Tabela 4.1: Classes relacionadas à leitura de dados do sistema.

Nome da Classe	Descrição da Classe
SensorCaminhaoData	Classe responsável pela leitura da existência de caminhão.
SensorTipoCaminhaoData	Classe responsável pela leitura do tipo de caminhão.
SensorGanchoData	Classe responsável pela leitura do estado do gancho do guindaste.
SensorRotacionadorData	Classe responsável pela leitura do rotacionador do guindaste.
SensorCaboData	Classe responsável pela leitura do cabo do guindaste.
SensorExisteCanaNoFosso	Classe responsável pela leitura da existência de cana no fosso.
SensorNFeixesCaminhao	Classe responsável pela leitura da quantidade de feixes de cana-de-açúcar.
SensorPesoMesa1	Classe responsável pela leitura do peso da mesa 1.
SensorPesoMesa2	Classe responsável pela leitura do peso da mesa 2.
SensorVolumeCaminhao	Classe responsável pela leitura do volume do caminhão.

Tabela 4.2: Classes relacionadas à escrita de dados do sistema.

Nome da Classe	Descrição da Classe
AtuadorGanchoData	Classe responsável pela atuação do gancho do guindaste.
AtuadorRotacionadorData	Classe responsável pela atuação do rotacionador do caminhão.
AtuadorCaboData	Classe responsável pela atuação do cabo do guindaste.
AtuadorCacambaData	Classe responsável pela atuação da caçamba do caminhão.
AtuadorChuveiro1Data	Classe responsável pela atuação do chuveiro da mesa 1.
AtuadorChuveiro2Data	Classe responsável pela atuação do chuveiro da mesa 2.
AtuadorVelocidadeMesa1Data	Classe responsável pela atuação da velocidade da mesa 1.
AtuadorVelocidadeMesa2Data	Classe responsável pela atuação da velocidade da mesa 2.

Abaixo, o esquema do sensor responsável pela existência de caminhão em *Object-Z*.

```
SensorCaminhaoData
AbstractData[BOOLEAN]
  INIT
  DataID = SENSOR_CAMINHAO_EXISTE
  DataName = "SensorCaminhao"
  InfoType = BOOLEAN
  DataUnits = NONE
  Type = READ_TYPE
  DataRate = HIGH_TYPE
```

Os demais esquemas dos sensores encontram-se no apêndice A.

Abaixo, as classes responsáveis pelo atuador do gancho do guindaste.

```
AtuadorGanchoData
AbstractData[BOOLEAN]
  INIT
  DataID = ATUADOR_GANCHO
  DataName = "AtuadorGancho"
  InfoType = BOOLEAN
  DataUnits = NONE
  Type = READ_TYPE
  DataRate = HIGH_TYPE
```

Os demais esquemas dos atuadores encontram-se no apêndice A.

4.2.1 Classes responsáveis pelo carregamento da cana-de-açúcar

Para se executar as comunicações com o meio externo, são utilizados os seguintes tipos para passagem de valores para canais especiais do tipo atuador ou sensores (MAHONY; DONG, 1999), referentes ao comando de:

Detecção de guindaste livre:

GUINDASTE_TYPE ::= GUINDASTE_LIVRE

Tipo de caminhão:

CAMINHAO_TYPE ::= CACAMBA | CORRENTE

Detecção do estado do cabo:

CABO_TYPE ::= CABO_DESCE
| CABO_SOBE

Detecção de gancho de guindaste livre:

$$GANCHO_TYPE ::= GANCHO_PRESO \mid GANCHO_SOLTO$$

Atuador do gancho do guindaste:

$$ROTACIONA_TYPE :== ROT_LADO \mid ROT_FRENTE \mid ROT_TRAS$$

Atuador da caçamba do caminhão:

$$CAMINHAO_CACAMBA_ACT_TYPE ::= NONE \mid ABRE_PORTA \mid DESPEJA_CANA \mid FECHA_PORTA$$

e atuador da mesa de carregamento:

$$MESA_ACT_TYPE ::= RUN \mid STOP$$

O processo de descarregamento de cana-de-açúcar na usina ocorre através da informação ao sistema da chegada de um novo caminhão, bem como o seu volume, e aguarda ordem de descarregamento na mesa ou no fosso. O percurso da entrada da usina até a mesa ou fosso tem a duração de um tempo K_{θ_C1} ou K_{θ_C1}, respectivamente.

Para o caso de caminhão de caçamba basculante, após o posicionamento correto, é solicitado o posicionamento do respectivo guindaste, a lateral da caçamba é aberta, o gancho é preso e então a cana-de-açúcar é despejada.

Pode-se perceber que a comunicação interprocessos do caminhão, mesa e fosso é realizada através dos canais, utilizando-se os tipos definidos acima.

CaminhaoCacambaProcess

$K_{\theta_C1} : \mathbb{T}$
$K_{\theta_C2} : \mathbb{T}$

$K_{\theta_C1} : 300$
$K_{\theta_C2} : 250$

$mesa2channel$: **chan**; $MESA_ACT_TYPE$
$fossochannel$: **chan**; $ACTION_PROCESS$
$mesa2InterfaceChannel$: **chan**; $MESA_ACT_TYPE$
$guindasteChannel$: **chan**; $GUINDASTE_TYPE$
$tipoCaminhaoChannel$: **chan**; $CAMINHAO_TYPE$
$ganchochannel$: **chan**; $GANCHO_TYPE$
$caminhaoActuator$: $CAMINHAO_CACAMBA_ACT_TYPE$ **actuator**
$dataBus$: $DataBus$
$detector$: $BOOLEAN$

INIT
$caminhaoActuator = NONE$
$detector = \text{false}$

$GetDetectorCaminhao$
$\Delta(detector)$
$detector' = dataBus.get(SENSOR_CAMINHAO_EXISTE)$

$RunGuindaste \mathrel{\widehat=} tipoCaminhaoChannel!CACAMBA \rightarrow$
$mesa2InterfaceChannel!RUN \rightarrow$
$guindastechannel?GUINDASTE_LIVRE \rightarrow$
$caminhaoActuator := ABRE_PORTA \rightarrow$
$ganchochannel?GANCHO_PRESO \rightarrow$
$caminhaoActuator := DESPEJA_CANA \rightarrow$
$ganchochannel?GANCHO_SOLTO \rightarrow$
$caminhaoActuator := FECHA_PORTA$
$guindastechannel!GUINDASTE_LIVRE$
$RunProcess \mathrel{\widehat=}= \mu M \bullet GetDetectorCaminhao \bullet [detector = \text{true}] \rightarrow$
$(mesa2channel?RUN \rightarrow$
$\text{WAITUNTIL}\ K_{\theta_C1} \rightarrow$
$RunGuindaste \rightarrow$
$mesa2InterfaceChannel?STOP \rightarrow$
$mesa2channel!STOP)$
$\overleftarrow{\Box}$
$(fossochannel?RUN \rightarrow$
$\text{WAITUNTIL}\ K_{\theta_C2} \rightarrow$
$RunGuindaste \rightarrow fossochannel!STOP)$
$\rightarrow M$

Para o caso de caminhão de correntes, são inseridos o valor do volume de cana-de-açúcar e o número de feixes e conseqüentemente calculado o volume de cada feixe, ocorrendo logo em seguida o seu descarregamento.

Sendo n_{fCC} o número de feixes, V_C o volume de cana no caminhão, V_{fCC} o volume de cada feixe, calculado através de n_{fCC} e V_C. O processo de descarregamento termina quando todos os feixes forem retirados.

CaminhaoCorrenteProcess

$K_{\theta_C1} : \mathbb{T}$
$K_{\theta_C2} : \mathbb{T}$

$K_{\theta_C1} : 300$
$K_{\theta_C2} : 250$

---

mesa2channel : **chan**; *MESA_ACT_TYPE*
fossochannel : **chan**; *ACTION_PROCESS*
mesa2InterfaceChannel : **chan**; *MESA_ACT_TYPE*
guindasteChannel : **chan**; *GUINDASTE_TYPE*
tipoCaminhaoChannel : **chan**; *CAMINHAO_TYPE*
ganchochannel : **chan**; *GANCHO_TYPE*
caminhaoActuator : *CAMINHAO_CACAMBA_ACT_TYPE* **actuator**
dataBus : *DataBus*
$n_{fCC} : \mathbb{N}$
$V_{fCC} : \mathbb{N}$
$V_C : \mathbb{N}$
detector : *BOOLEAN*

INIT

caminhaoActuator = *NONE*
$n_{fCC} = 0$
$V_{fCC} = 0$
$V_C = 0$

GetDetectorCaminhao

$\Delta(detector)$

$detector' = dataBus.get(SENSOR_CAMINHAO_EXISTE)$
$n'_{fCC} = dataBus.get(SENSOR_NFEIXES_CAMINHAO)$
$V'_{fCC} = dataBus.get(SENSOR_VOLUME_FEIXES)$

DecrementaFeixes

$\Delta(n_{fCC})$

$n_{fCC} > 0$
$n'_{fCC} = n_{fCC} - 1$

$RunGuindaste \mathrel{\widehat=} \mu F\ tipoCaminhaoChannel!CORRENTE \rightarrow$
$mesa2InterfaceChannel!RUN \rightarrow$
$guindasteChannel?GUINDASTE_LIVRE \rightarrow$
$ganchochannel?GANCHO_PRESO \rightarrow$
$ganchochannel?GANCHO_SOLTO \rightarrow$
$guindasteChannel!GUINDASTE_LIVRE \rightarrow$
$DecrementaFeixes \bullet [n_{fCC} > 0] \rightarrow F$

$$
\begin{aligned}
& RunProcess \mathrel{\widehat=} \mu M \bullet GetDetectorCaminhao \bullet [detector = \text{true}] \rightarrow \\
& \quad (mesa2channel?RUN \rightarrow \text{WAITUNTIL}\ K_{\theta_C1} \rightarrow \\
& \qquad RunGuindaste \rightarrow \\
& \qquad mesa2InterfaceChannel?STOP \rightarrow \\
& \qquad mesa2channel!STOP) \\
& \quad \overleftarrow{\Box} \\
& \quad (fossochannel?RUN \rightarrow \text{WAITUNTIL}\ K_{\theta_C2} \rightarrow \\
& \qquad RunGuindaste \rightarrow fossochannel!STOP) \\
& \rightarrow M
\end{aligned}
$$

A classe guindaste modela as seqüências de movimentos necessários para se efetuar o descarregamento, tanto para caminhões com caçamba basculante como também para caminhões com corrente.

Para cada um dos tipos de caminhões, existe uma seqüência pré-determinada de eventos que devem ser seguidos, além da necessidade de troca da garra de acordo com o tipo de caminhão. Pode-se perceber que a classe *Guindaste* possui os atuadores *rotacionador*, *gancho* e *cabo*.

GuindasteProcess

$$
\begin{aligned}
& guindasteChannel : \mathbf{chan};\ GUINDASTE_TYPE \\
& tipoCaminhaoChannel : \mathbf{chan};\ CAMINHAO_TYPE \\
& ganchochannel : \mathbf{chan};\ GANCHO_TYPE \\
& rotacionador : ROTACIONA_TYPE\ \mathbf{actuator} \\
& gancho : GANCHO_TYPE\ \mathbf{actuator} \\
& cabo : CABO_TYPE\ \mathbf{actuator} \\
& garracacamba : BOOLEAN \\
& garracorrente : BOOLEAN \\
\hline
& garracacamba = \text{true} \bullet garracorrente = \text{false} \\
& garracacamba = \text{false} \bullet garracorrente = \text{true}
\end{aligned}
$$

INIT

$$
\begin{aligned}
& rotacionador = \text{false} \\
& gancho = GANCHO_SOLTO \\
& cabo = CABO_SOBE \\
& garracacamba = \text{true}
\end{aligned}
$$

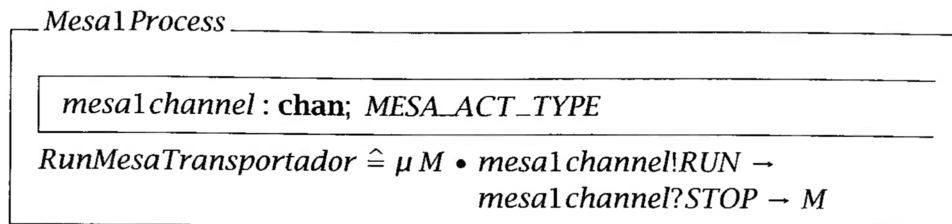
TrocaGarraCorrenteCacamba

$$
\begin{aligned}
& \Delta(garracacamba, garracorrente) \\
& tipo? : GUINDASTE_TYPE \\
\hline
& tipo? = GUINDASTE_CACAMBA \bullet garracacamba = false \bullet \\
& \qquad garracacamba' = true \\
& tipo? = GUINDASTE_CORRENTE \bullet garracorrente = false \bullet \\
& \qquad garracorrente' = true
\end{aligned}
$$

$$
\begin{array}{l}
RunComCacamba \mathrel{\widehat=} garracacamba = \text{true} \bullet \\
\quad (rotacionador := ROT_FRENTE \rightarrow \\
\quad cabo := CABO_DESCE \rightarrow \\
\quad gancho := GANCHO_PRESO \rightarrow \\
\quad ganchoChannel!GANCHO_PRESO \rightarrow \\
\quad cabo := CABO_SOBE \rightarrow \\
\quad rotacionador := ROT_TRAS \rightarrow \\
\quad gancho := GANCHO_SOLTO \rightarrow \\
\quad ganchoChannel!GANCHO_SOLTO) \\
RunComCorrente \mathrel{\widehat=} garracacorrente = \text{true} \bullet \\
\quad (rotacionador := ROT_FRENTE \rightarrow \\
\quad rotacionador := ROT_LADO \rightarrow \\
\quad cabo := CABO_DESCE \rightarrow \\
\quad gancho := GANCHO_PRESO \rightarrow \\
\quad ganchoChannel!GANCHO_PRESO \rightarrow \\
\quad cabo := CABO_SOBE \rightarrow \\
\quad rotacionador := ROT_FRENTE \rightarrow \\
\quad cabo := CABO_DESCE \rightarrow \\
\quad gancho := GANCHO_SOLTO \rightarrow \\
\quad ganchoChannel!GANCHO_SOLTO \rightarrow \\
\quad cabo := CABO_SOBE \rightarrow \\
\quad rotacionador := ROT_LADO \rightarrow \\
\quad rotacionador := ROT_TRAS) \\
RunGuindaste \mathrel{\widehat=} \mu\, G \bullet [t : CAMINHAO_TYPE] \bullet \\
\quad tipoCaminhaoChannel?t \rightarrow \\
\quad TrocaGarraCorrenteCacamba; \\
\quad ([t = CACAMBA] \bullet \\
\qquad guindasteChannel!GUINDASTE_LIVRE \rightarrow \\
\qquad RunComCacamba \rightarrow \\
\qquad guindasteChannel?GUINDASTE_LIVRE \rightarrow \\
\qquad G) \\
\quad \Box \\
\quad ([t = CORRENTE] \bullet \\
\qquad guindasteChannel!GUINDASTE_LIVRE \rightarrow \\
\qquad RunComCorrente \rightarrow \\
\qquad guindasteChannel?GUINDASTE_LIVRE \rightarrow \\
\qquad G)
\end{array}
$$

Para a modelagem e especificação da mesa foram utilizados dois processos independentes, o primeiro relacionado com a sincronização das mesas em relação ao processo de descarregamento dos caminhões e do fosso, e o segundo, relacionado com o funcionamento da mesa (esteira e chuveiro), de modo independente em relação ao funcionamento do processo de descarregamento da cana-de-açúcar, especificado em 4.2.2.

Para o processo de carregamento de cana-de-açúcar para a produção, existem duas mesas responsáveis, onde a mesa número 1 é responsável pelo carregamento da cana-de-açúcar proveniente do fosso, enquanto que a mesa número 2 é responsável pelo carregamento proveniente dos caminhões.



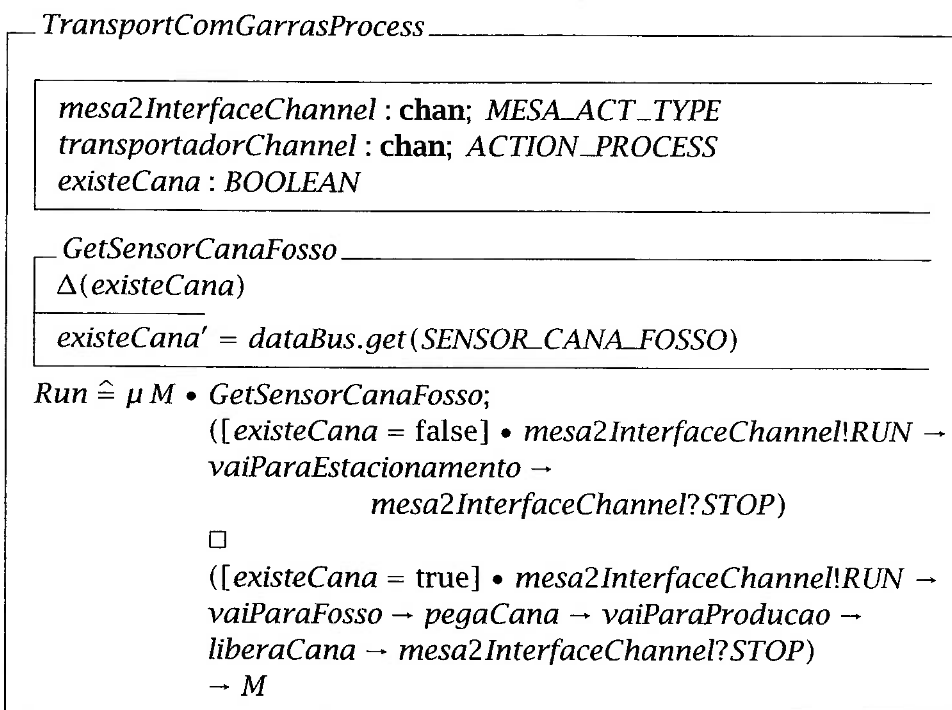
Mesa1Process

$mesa1channel : \textbf{chan};\ MESA_ACT_TYPE$

$RunMesaTransportador \mathrel{\widehat=} \mu M \bullet mesa1channel!RUN \rightarrow mesa1channel?STOP \rightarrow M$

A mesa número 1 é sincronizada em relação à classe *TransportComGarras* através do canal *mesa1channel*.

A classe *TransportComGarras* por sua vez, possui sensores notificando a existência de cana-de-açúcar, e caso o fosso esteja vazio, o transportador com garras permanece aguardando no estacionamento.



TransportComGarrasProcess

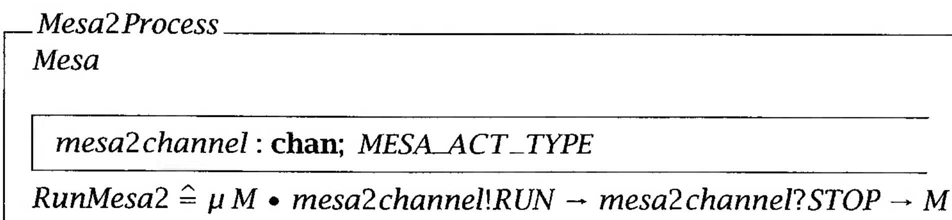
$mesa2InterfaceChannel : \textbf{chan};\ MESA_ACT_TYPE$
$transportadorChannel : \textbf{chan};\ ACTION_PROCESS$
$existeCana : BOOLEAN$

GetSensorCanaFosso
$\Delta(existeCana)$

$existeCana' = dataBus.get(SENSOR_CANA_FOSSO)$

$Run \mathrel{\widehat=} \mu M \bullet GetSensorCanaFosso;$
$([existeCana = \text{false}] \bullet mesa2InterfaceChannel!RUN \rightarrow$
$vaiParaEstacionamento \rightarrow$
$mesa2InterfaceChannel?STOP)$
$\Box$
$([existeCana = \text{true}] \bullet mesa2InterfaceChannel!RUN \rightarrow$
$vaiParaFosso \rightarrow pegaCana \rightarrow vaiParaProducao \rightarrow$
$liberaCana \rightarrow mesa2InterfaceChannel?STOP)$
$\rightarrow M$

A mesa número 2 é sincronizada com as classes responsáveis pelo caminhão através do canal *mesa2channel*.

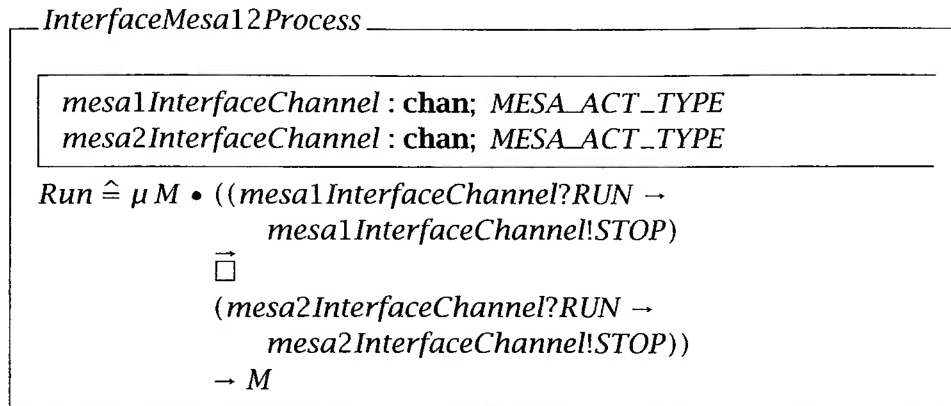


Mesa2Process
Mesa

$mesa2channel : \textbf{chan};\ MESA_ACT_TYPE$

$RunMesa2 \mathrel{\widehat=} \mu M \bullet mesa2channel!RUN \rightarrow mesa2channel?STOP \rightarrow M$

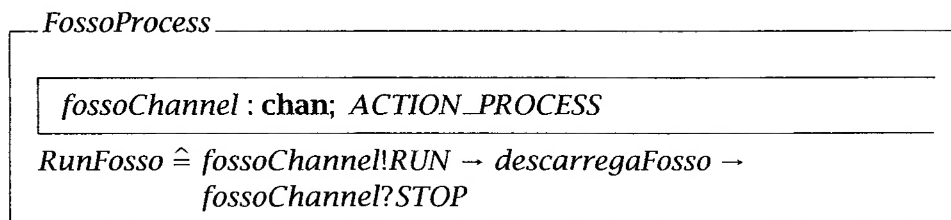
A classe *InterfaceMesa12* somente é responsável pela sincronização das mesas e pela definição das prioridades existentes entre elas. Pode-se perceber que o processo responsável pela mesa 2 possui uma prioridade maior do que o da mesa número 1.



```
InterfaceMesa12Process
  mesa1InterfaceChannel : chan; MESA_ACT_TYPE
  mesa2InterfaceChannel : chan; MESA_ACT_TYPE

  Run ≙ μ M • ((mesa1InterfaceChannel?RUN →
                  mesa1InterfaceChannel!STOP)
               □→
               (mesa2InterfaceChannel?RUN →
                  mesa2InterfaceChannel!STOP))
               → M
```

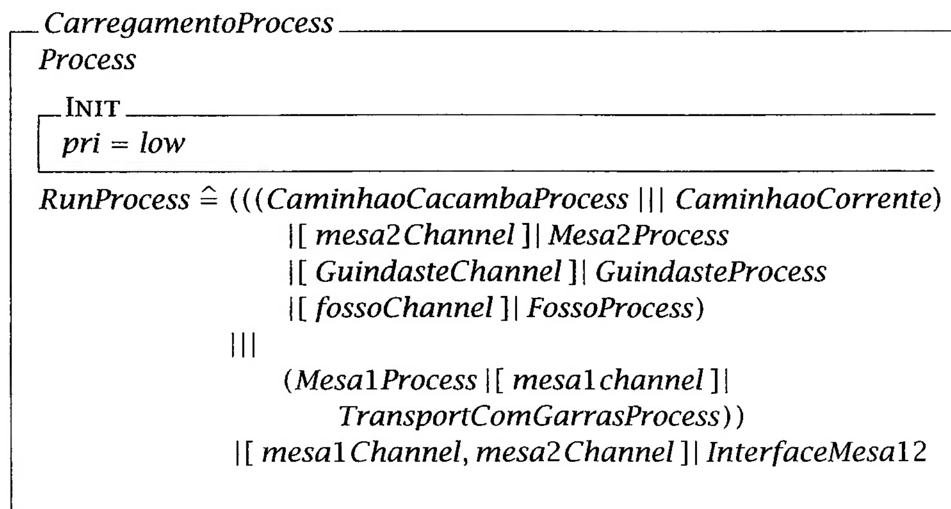
O Fosso por sua vez, recebe o carregamento do caminhão de correntes ou caçamba basculante, caso a mesa número 2 esteja ocupada. Pode-se perceber que a classe *Mesa* possui uma prioridade maior do que a classe *Fosso*, uma vez que todos os caminhões estão modelados e especificados de modo a escolher sempre a mesa, caso as duas opções (mesa ou fosso) estejam disponíveis.



```
FossoProcess
  fossoChannel : chan; ACTION_PROCESS

  RunFosso ≙ fossoChannel!RUN → descarregaFosso →
             fossoChannel?STOP
```

O processo de carregamento pode ser modelado através dos seguintes processos, funcionando paralelamente.



```
CarregamentoProcess
  Process
  INIT
    pri = low

  RunProcess ≙ (((CaminhaoCacambaProcess ||| CaminhaoCorrente)
                  |[ mesa2Channel ]| Mesa2Process
                  |[ GuindasteChannel ]| GuindasteProcess
                  |[ fossoChannel ]| FossoProcess)
               |||
                  (Mesa1Process |[ mesa1channel ]|
                      TransportComGarrasProcess))
               |[ mesa1Channel, mesa2Channel ]| InterfaceMesa12
```

4.2.2 Especificação de Eventos Assíncronos

A classe *EventosProcess*, responsável pela notificação de eventos externos provenientes da interface homem-máquina é também responsável pelo funcionamento da esteira que leva a cana-de-açúcar à produção, sendo que ela possui como atuador

um chuveiro e pode funcionar em três níveis diferentes de velocidade. Existem na usina, duas mesas transportadoras, sendo que a mesa 1 é responsável pela cana proveniente do fosso e a mesa 2 recebe a cana diretamente dos caminhões.

Cada uma das mesas pode funcionar em três diferentes velocidades, possuindo interface para aumentar ou diminuir a velocidade das mesmas.

Quando o volume total de cana acumulado na Mesa excede um determinado valor limite, tem-se uma situação de falha. Além disto, o sistema supervisório pode interromper o funcionamento da mesa a qualquer instante, desligando a esteira e o chuveiro.

Pode-se perceber a utilização da uma variável *stopFlag*, para sinalizar a requisição do sistema supervisório para a parada da esteira. A sua implementação poderia ser efetuada através da utilização do operador de interrupção $\triangledown$ (HOARE, 1985), mas a sua especificação não foi utilizada uma vez que este operador não é implementado em nenhuma linguagem ou biblioteca baseada em CSP.

Assim, a classe *EventosProcess* é responsável pela interface com o usuário para efetuar a parada da esteira e aumentar ou diminuir a velocidade da esteira.

Abaixo, são enumerados os tipos de dados utilizados na especificação de eventos assíncronos, e utilizados nos diversos canais de sincronização que a compõem.

O canal de controle da velocidade da mesa descreve as 4 possíveis velocidades que esta pode operar.

$$
\begin{aligned}
VELOCIDADE_MESA_TYPE ::= \; & V_MESA_DESLIGADO \\
| \; & V1_MESA \mid V2_MESA \mid V3_MESA
\end{aligned}
$$

No caso dos eventos assíncronos que a compõem, pode-se separá-los em dos tipos principais, os relacionados com o controle da velocidade da mesa e as relacionadas com eventos de emergência da mesa.

$$
\begin{aligned}
EVENTOS_CHANNEL_TYPE ::= \; & VELOCIDADE_CHANNEL_TYPE \\
| \; & EMERGENCIA_TYPE
\end{aligned}
$$

Em relação ao controle de velocidade, pode-se listar apenas os eventos relacionados com o aumento ou diminuição da velocidade.

$$
\begin{aligned}
VELOCIDADE_CHANNEL_TYPE ::= \; & V_AUMENTA_TYPE \\
| \; & V_DIMINUI_TYPE
\end{aligned}
$$

Em relação à emergência, apenas os eventos de parada ou início de operação da mesa são enumerados logo abaixo.

$$
EMERGENCIA_TYPE ::= STOP \mid RUN
$$

A seguir, é enumerado o processo relacionado ao tratamento de eventos assíncronos. No caso, pode-se perceber que as comunicações com outros processos externos são efetuadas através dos canais *chuveiro*, *esteira*, *variadorVelocidade*, *velocidadeChannel* e *supervisorioChannel*.

EventosProcess

Process

$WAIT_TIME : \mathbb{T}$

$WAIT_TIME : 50$

$stopFlag : BOOLEAN$
$velocidadeAtual : VELOCIDADE_MESA_TYPE$
$pesoMesa : DOUBLE$
$chuveiro : BOOLEAN$ **actuator**
$esteira : BOOLEAN$ **actuator**
$variadorVelocidade : VELOCIDADE_MESA_TYPE$ **actuator**
$velocidadeChannel :$ **chan**; $VELOCIDADE_CHANNEL_TYPE$
$supervisorioChannel :$ **chan**; $ACTION_PROCESS$

INIT

$pri = lowest$
$stopFlag = \mathrm{true}$
$velocidadeAtual = V_MESA_DESLIGADO$
$pesoMesa = 0.0;$

aumentaVelocidade

$\Delta velocidadeAtual$

$velocidadeAtual = V_MESA_DESLIGADO \bullet velocidadeAtual' = V1_MESA$
$velocidadeAtual = V1_MESA \bullet velocidadeAtual' = V2_MESA$
$velocidadeAtual = V2_MESA \bullet velocidadeAtual' = V3_MESA$
$variadorVelocidade := velocidadeAtual'$

diminuiVelocidade

$\Delta velocidadeAtual$

$velocidadeAtual = V3_MESA \bullet velocidadeAtual' = V2_MESA$
$velocidadeAtual = V2_MESA \bullet velocidadeAtual' = V1_MESA$
$variadorVelocidade := velocidadeAtual'$

GetPesoMesa

$\Delta(pesoMesa)$

$pesoMesa' = dataBus.get(SENSOR_PESO_MESA_1)$

setStopFlag

$\Delta stopFlag$
$flagValue? : BOOLEAN$

$stopFlag' = flagValue?$

$$
\begin{array}{l}
RunMesa \mathrel{\widehat=} GetPesoMesa \rightarrow \\
\qquad (([pesoMesa > 0.0 \land stopFlag = \text{false}] \bullet \\
\qquad\quad chuveiro1 := true \rightarrow esteira1 := true) \\
\qquad \Box \\
\qquad ([pesoMesa = 0.0 \lor stopFlag = \text{true}] \bullet \\
\qquad\quad chuveiro1 := false \rightarrow esteira1 := false)) \\
RunProcess \mathrel{\widehat=} (\\
\qquad eventosChannel?v \rightarrow \\
\qquad ([v : VELOCIDADE_CHANNEL_TYPE] \bullet \\
\qquad\quad velocidadeChannel?v \rightarrow \\
\qquad\quad ([v = V_AUMENTA_TYPE] \bullet \\
\qquad\qquad aumentaVelocidade \\
\qquad\quad [v = V_DIMINUI_TYPE] \bullet \\
\qquad\qquad diminuiVelocidade)) \\
\qquad \lor \\
\qquad ([v : EMERGENCIA_TYPE] \bullet \\
\qquad\quad velocidadeChannel?v \rightarrow \\
\qquad\quad v = STOP \bullet (supervisorioChannel?STOP \rightarrow \\
\qquad\qquad setStopFlag(\text{false})) \\
\qquad\quad \lor \\
\qquad\quad v = RUN \bullet (supervisorioChannel?RUN \rightarrow \\
\qquad\qquad setStopFlag(\text{true}))) \\
\quad \Box \\
\qquad (\textsc{Wait}\ WAIT_TIME \rightarrow \textsc{Skip}) \\
\quad) \\
\quad \rightarrow RunMesa
\end{array}
$$

Os processos acima, por sua vez, são escalonados pela classe *Scheduler* em um processo de menor prioridade. Periodicamente, o processo verifica a existência ou não da ocorrência de algum evento, durante um determinado período de tempo, definido pela constante *WAIT_TIME*, e caso o mesmo não ocorra, o processo termina, através do operador CSP SKIP.

4.2.3 Classe principal

A classe principal é modelada logo abaixo, em que são instanciados os principais processos e são definidos os seus estados iniciais, no caso, as classes *Scheduler*, que é responsável pelo disparo das classes *DataListener*, *CarregamentoProcess* e *EventosProcess*, além da classe *SecurityReport* que é executada paralelamente à classe *Scheduler*.

Main

$scheduler : Scheduler$
$highProcess : DataListener$
$lowProcess : CarregamentoProcess$
$lowestProcess : EventosProcess$
$syncHighProcess : SyncProcess$
$syncLowProcess : SyncProcess$
$syncLowestProcess : SyncProcess$
$SecurityReport : SecurityReportProcess$
$ownerOfThread : ID \rightarrowtail processes$
$processes : PROCESS \times PRIORITY$
$syncChannelArray : Array[\mathbf{chan}; ACTION_PROCESS]$
$runChannelArray : Array[\mathbf{chan}; ACTION_PROCESS]$
$securityChannelArray : Array[\mathbf{chan}; ACTION_PROCESS]$
$dataBus : DataBus$

Classe principal do sistema

INIT

$processes = \varnothing$

Inicialização da classe DataBus

$dataBus = DataBus.GetDataBusInstance$

addProcess

$\Delta processes$
$process? : PROCESS$
$priority? : PRIORITY$

$processes \cap process? = \varnothing$
$processes' = processes \cup \{(process?, priority?)\}$

removeProcess

$\Delta processes$
$index? : ID$

$processes' = processes \setminus ownerOfThread(index?)$

$$
\begin{aligned}
Main \mathrel{\widehat=}\ & scheduler \,|[\, syncChannelArray \,]|\\
& ((syncHighProcess \,|[\, runChanelArray \,]|\, highProcess)\\
& \parallel (syncLowProcess \,|[\, runChanelArray \,]|\, lowProcess)\\
& \parallel (syncLowestProcess\\
& \qquad |[\, runChanelArray \,]|\, lowestProcess))\\
& |[\, securityChannelArray \,]|\, WatchDogProcess
\end{aligned}
$$

Na especificação acima, pode-se perceber o canal de sincronização existente entre cada um dos processos através do operados de paralelismo $A \,|[\, X \,]|\, B$, sendo os processos A e B sincronizados através do canal X.

4.3 Validação por *traces*

Através do refinamento de *traces*, pode-se verificar a especificação do sistema através da sua validação, ou seja, um processo $P1$ atende às especificações de $P0$ se para cada *trace* de $P1$ é permitida por $P0$. Ou seja, $P1$ é considerado um refinamento de $P0$.

$$P0 \sqsubseteq_T P1 = traces(P1) \subseteq traces(P0)$$

Após a definição dos refinamentos do sistema, a sua validação pode ser mecanizada através da utilização de checadores de modelo como FDR (GOLDSMITH, 2001). A sua utilização envolve a transcrição do modelo CSP-OZ para a linguagem CSP_M (*Machine Readable CSP*) e são verificados através de *assertions* (KASSEL; SMITH, 2001; MOTA; SAMPAIO, 2001; MOTA; BORBA; SAMPAIO, 2002).

4.3.1 Refinamento da classe *GuindasteProcess*

$$\begin{aligned}
RunComCacamba \mathrel{\widehat=} & rotacionador := ROT_FRENTE \rightarrow \\
& cabo := CABO_DESCE \rightarrow gancho := GANCHO_PRESO \rightarrow \\
& ganchoChannel!GANCHO_PRESO \rightarrow cabo := CABO_SOBE \rightarrow \\
& rotacionador := ROT_TRAS \rightarrow gancho := GANCHO_SOLTO \rightarrow \\
& ganchoChannel!GANCHO_SOLTO \\
RunComCorrente \mathrel{\widehat=} & rotacionador := ROT_FRENTE \rightarrow \\
& rotacionador := ROT_LADO \rightarrow cabo := CABO_DESCE \rightarrow \\
& gancho := GANCHO_PRESO \rightarrow ganchoChannel!GANCHO_PRESO \rightarrow \\
& cabo := CABO_SOBE \rightarrow rotacionador := ROT_FRENTE \rightarrow \\
& cabo := CABO_DESCE \rightarrow gancho := GANCHO_SOLTO \rightarrow \\
& ganchoChannel!GANCHO_SOLTO \rightarrow cabo := CABO_SOBE \rightarrow \\
& rotacionador := ROT_LADO \rightarrow rotacionador := ROT_TRAS \\
GuindasteProcessTrace \mathrel{\widehat=} & RunComCacamba \,\Box\, RunComCorrente
\end{aligned}$$

$$Main \sqsubseteq_T GuindasteProcessTrace$$

4.3.2 Refinamento da classe *CaminhaoCacambaProcess*

$$\begin{aligned}
CaminhaoCacambaProcessTraces \mathrel{\widehat=} & guindastechannel?GUINDASTE_LIVRE \rightarrow \\
& caminhaoActuator := ABRE_PORTA \rightarrow \\
& ganchochannel?GANCHO_PRESO \rightarrow \\
& caminhaoActuator := DESPEJA_CANA \rightarrow \\
& ganchochannel?GANCHO_SOLTO \rightarrow \\
& caminhaoActuator := FECHA_PORTA \\
& guindastechannel!GUINDASTE_LIVRE
\end{aligned}$$

$$Main \sqsubseteq_T CaminhaoCacambaProcessTraces$$

4.3.3 Refinamento da classe *CaminhaoCorrentesProcess*

$$CaminhaoCorrentesProcessTraces \mathrel{\widehat=} guindasteChannel?GUINDASTE_LIVRE \rightarrow ganchochannel?GANCHO_PRESO \rightarrow ganchochannel?GANCHO_SOLTO \rightarrow guindasteChannel!GUINDASTE_LIVRE$$

$$Main \sqsubseteq_T CaminhaoCorrentesProcessTraces$$

4.3.4 Refinamento da classe *Mesa1Process*

$$Mesa1ProcessTraces \mathrel{\widehat=} mesa1channel!RUN \rightarrow mesa1channel!STOP$$

$$Main \sqsubseteq_T Mesa1ProcessTraces$$

4.3.5 Refinamento da classe *TransporteComGarrasProcess*

$$TransportComGarrasProcessTraces \mathrel{\widehat=} vaiParaEstacionamento \;\Box\; (vaiParaFosso \rightarrow pegaCana \rightarrow vaiParaProducao \rightarrow liberaCana)$$

$$Main \sqsubseteq_T TransportComGarrasProcessTraces$$

4.3.6 Refinamento da classe *Mesa2Process*

$$Mesa2ProcessTraces \mathrel{\widehat=} mesa2channel!RUNthenmesa2channel!STOP$$

$$Main \sqsubseteq_T Mesa2ProcessTraces$$

4.3.7 Refinamento da classe *FossoProcess*

$$FossoProcessTraces \mathrel{\widehat=} fossoChannel!RUNthendescarregaFosso \rightarrow fossoChannel!STOP$$

$$Main \sqsubseteq_T FossoProcess$$

5 Implementação e Simulação

Para a implementação de um simulador aderente às especificações propostas por CSP, pode-se destacar a linguagem de programação OCCAM (HOARE, 1988; WOOD; WELCH, 1996), na qual disponibiliza as implementações padrões para CSP e que serviram de base para implementações da álgebra CSP em linguagens mais populares como C++ e Java. Para a linguagem C++, a biblioteca que provê o modelo OCCAM são as bibliotecas desenvolvidas pela Universidade de Kent (*"The Kent C++ Library"*) (BROWN; WELCH, 2003), enquanto que para a linguagem Java, podemos destacar o desenvolvimento de JCSP (*"Java Communicating Sequential Processes"*) (P.D.AUSTIN; P.H.WELCH, 2000; P.H.WELCH; J.M.R.MARTIN, 2000) e CTJ (*"Communicating Threads for Java"*) (HILDERINK, 2000).

A biblioteca escolhida para a implementação do simulador foi JCSP devido ao suporte que a linguagem Java oferece para a programação de sistemas distribuídos e paralelos, interoperabilidade entre diversos sistemas operacionais, existência de gerenciamento automático de memória. Apesar dos problemas existentes para o desenvolvimento de sistemas de tempo real utilizando-se a linguagem Java, existem algumas propostas de especificações para a utilização da mesma na implementação e no desenvolvimento de sistemas em tempo real (BOLLELLA, 2000). As principais diferenças e semelhanças entre as bibliotecas CTJ e JCSP podem ser observadas em Schaller, Hilderink e Welch (2000), concluindo-se que ambas as bibliotecas atendem os requisitos propostos para se implementar a semântica definida por CSP.

Diversos trabalhos foram propostos para a implementação de software a partir da especificação CSP ou CSP-OZ (FISCHER, 2000; CAVALCANTI; SAMPAIO, 2002; NEVISON, 2001; P.H.WELCH; J.M.R.MARTIN, 2000). Neste capítulo, apresentaremos a arquitetura do sistema utilizado para se implementar os padrões de projeto estudado nos capítulos anteriores, bem como a implementação do estudo de caso, utilizando-se a biblioteca JCSP e descrevendo as suas principais características e as considerações efetuadas para se converter o modelo abstrato ao modelo real. Para uma melhor compreensão do sistema, foram utilizados diagramas informais e figuras, em particular diagramas de classe UML (RUMBAUGH; JACOBSON; BOOCH, 1998).

5.1 Estrutura

O desenvolvimento do sistema foi dividido em módulos, sendo que cada um dos módulos ficou responsável pela implementação de algumas tarefas. A interdependência dos módulos pode ser visualizada no diagrama de componentes da figura 5.1.

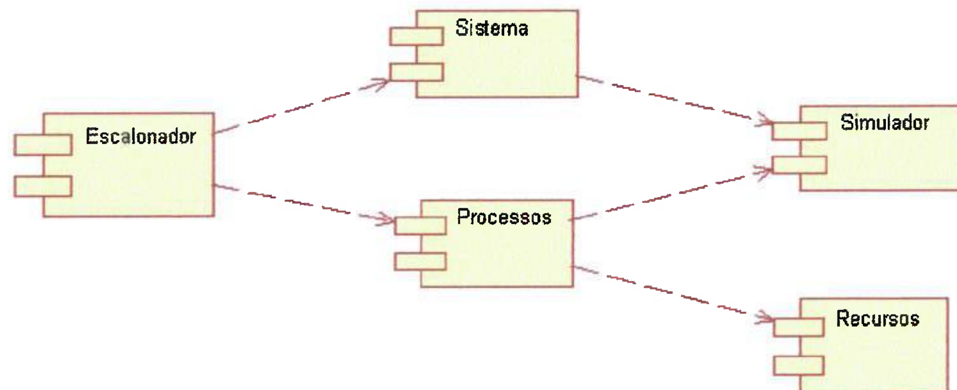


Figura 5.1: Diagrama de componentes.

- **Processos**: Definição das classes abstratas relacionada aos processos.
- **Recursos**: Definição das classes abstratas relacionada aos recursos utilizados.
- **Escalonador**: Definição das classes abstratas relacionada ao escalonador e ao programa principal.
- **Sistema**: Definição das classes relacionadas ao carregamento de cana-de-açúcar, do estudo de caso.
- **Simulador**: Definição das classes relacionadas à simulação dos dados provenientes dos sensores e atuadores.

Com relação aos dados de entrada e saída do estudo de caso, os seguintes recursos podem ser visualizados no diagrama de classes 5.2.

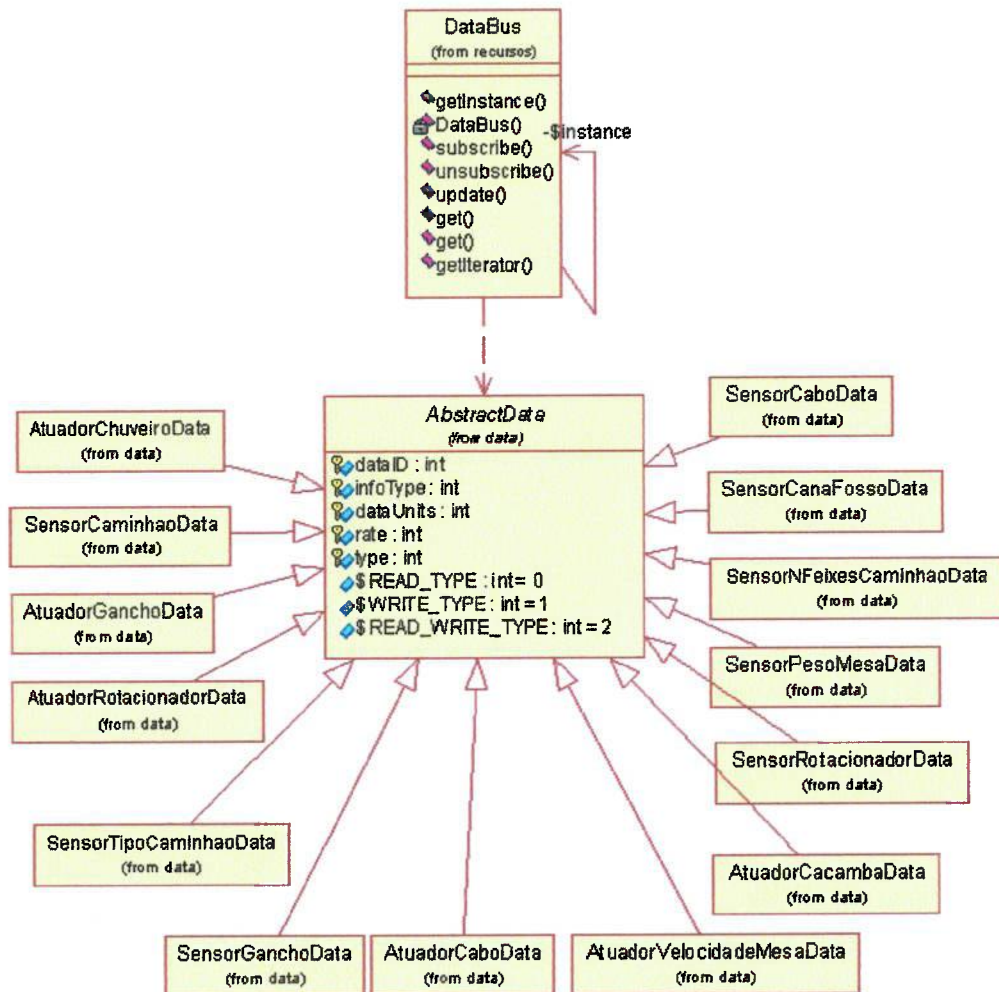


Figura 5.2: Dados de entrada e saída.

Pode-se observar que todas as classes relacionadas a recursos são derivadas da classe *AbstractData* e são armazenadas na classe *DataBus*.

Os processos que compõem o sistema podem ser visualizados pelo diagrama de classes da figura 5.3:

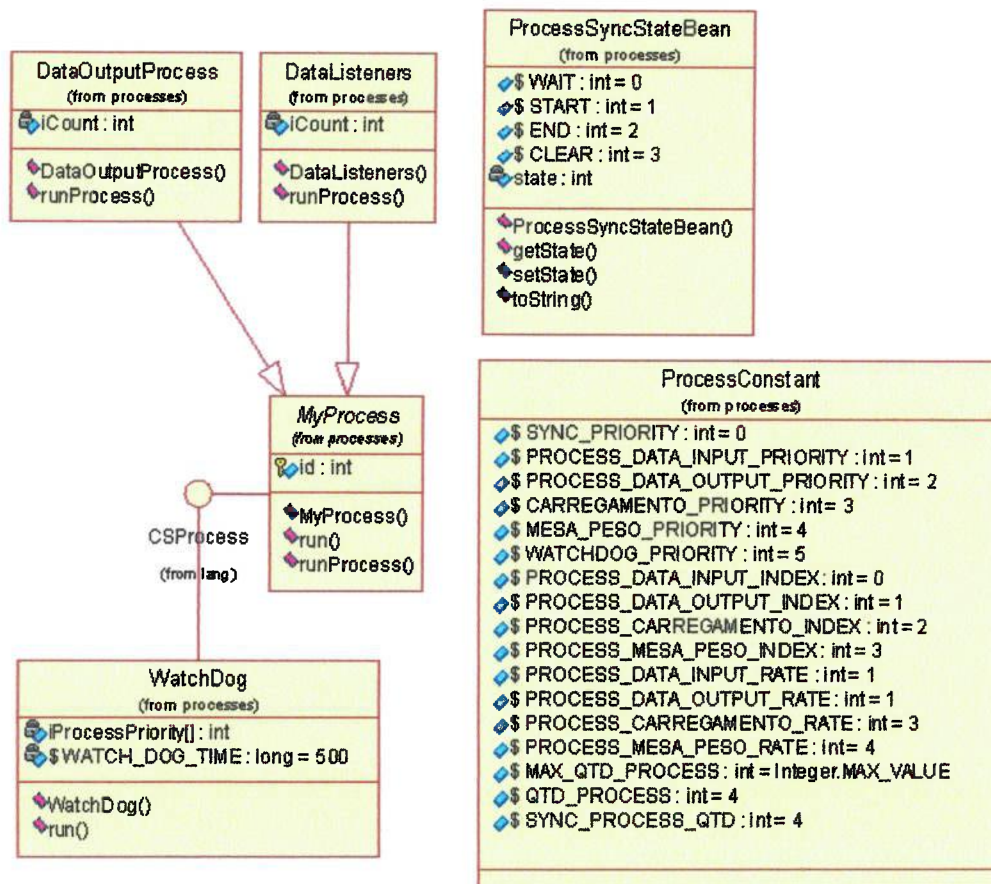


Figura 5.3: Processos básicos do sistema.

A figura 5.3 refere-se ao diagrama de classes dos processos básicos que compõem o sistema, ou seja, os processos de leitura e escrita de dados e o processo de monitoramento dos processos. A figura 5.4, por sua vez, descreve os processos que compõem o sistema de carregamento de cana-de-açúcar do estudo de caso.

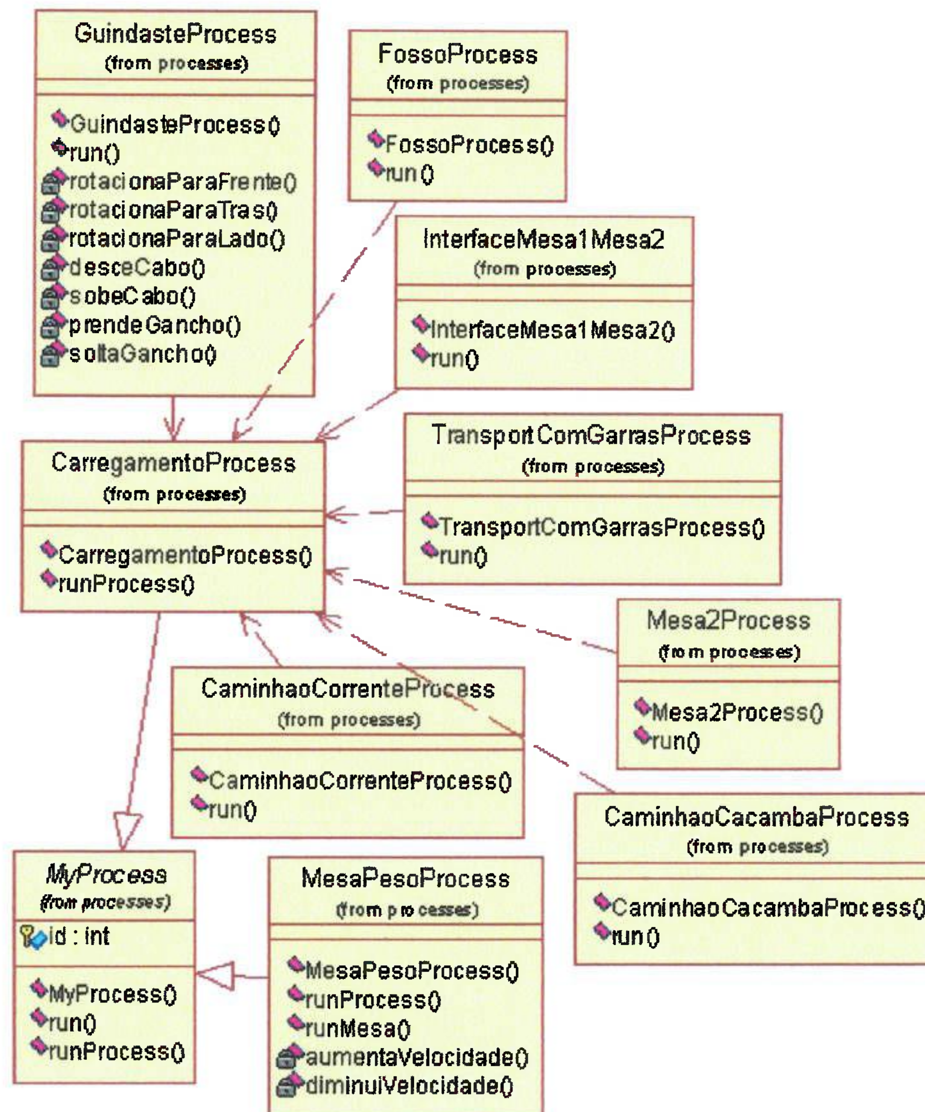


Figura 5.4: Processos do sistema de carregamento de cana-de-açúcar.

Ou seja, o processo de carregamento de cana-de-açúcar é composto de sub-processos inter-relacionados, no caso, dos processos relacionados ao guindaste, fosso, mesa 1, mesa 2, transportador com garras, caminhão com correntes e caminhão com caçamba basculante, sincronizados através da utilização de canais, descrito no capítulo 4.

5.2 Considerações particulares

Devido a algumas diferenças em relação ao modelo abstrato apresentado nos capítulos anteriores e ao modelo real, em particular a sua implementação como um simulador, são apresentadas abaixo as principais diferenças e as decisões tomadas na sua implementação.

Também são destacadas as diferenças existentes entre o modelo abstrato e o modelo real devido às particularidades existente na linguagem Java e na biblioteca JCSP.

5.2.1 Implementação de atuadores e sensores

Em relação à implementação da simulação dos atuadores e sensores, algumas considerações podem ser destacadas:

1. Os sensores foram simulados através de um banco de dados centralizados, a classe *DataListener* periodicamente consulta os valores inseridos no banco de dados.
2. A ordem dos valores do banco de dados segue a ordem da enumeração da variável ID_TYPE.
3. Os tipos inseridos correspondem à definição de tipos de cada uma das variáveis.
4. Apesar de todos os valores serem lidos ordenadamente por um único processo, no caso *DataOutputProcess* e *DataListener*, a leitura dos sensores poderia ser implementado por mais de um processo, com diferentes prioridades e taxas de leitura de acordo com a necessidade de atualização ou prioridades. No estudo de caso, por definição, todas as variáveis foram inseridas em apenas um processo de leitura e um processo de escrita.
5. Os canais dos atuadores não foram implementados, uma vez que os mesmos correspondem a eventos externos. Para visualizarmos a sua simulação, cada um dos atuadores escreve uma notificação na tela do usuário.

A tabela referente aos valores dos sensores são descritos na tabela 5.1:

Tabela 5.1: Tabela com dados de sensores simulados.

ID_TYPE	Sensor	Descrição
1	SensorCaminhaoData	Sensor de existencia de caminhão
2	SensorTipoCaminhaoData	Sensor de tipo de caminhão
3	SensorGanchoData	Sensor de gancho preso ou solto, 0 = solto, 1 = preso
4	SensorRotacionadorData	Sensor de rotacionador, 0 = lado, 1 = frente, 2 = trás
5	SensorCaboData	Sensor de cabo, 0 = Desce, 1 = sobe
6	SensorExisteCanaNoFosso	Sensor de cana no fosso, checado = true
7	SensorNFeixesCaminhao	Sensor de quantidade de feixes no caminhao de corrente
8	SensorPesoMesa1	Sensor de peso da mesa 1
9	SensorPesoMesa2	Sensor de peso da mesa 2
10	SensorVolumeCaminhao	Sensor de volume do caminhão

O simulador busca os valores de cada um dos sensores através da definição da variável *InfoType*, por exemplo, caso o tipo de variável seja um BOOLEAN ou inteiro, o simulador busca o seu valor na coluna adequada ou faz o tratamento do dado de acordo com o tipo definido.

5.2.2 Implementação de prioridades

Para a construção das prioridades de cada um dos processos, foi utilizada a classe *PriParallel* da biblioteca JCSP, na qual a maior prioridade de um processo é atribuída ao menor índice correspondente ao método construtor da classe. A semântica da prioridade no caso é a mesma implementada pela máquina virtual Java utilizada, e o seu limite de possíveis prioridades definidas pela mesma, na linguagem Kroc (*"The Kent retargetable OCCAM compiler"*) (WOOD; WELCH, 1996). A implementação de prioridades foi estendida para 32 possíveis níveis (BARNES; WELCH, 2002a, 2002b). Um exemplo de sua construção é descrito logo abaixo:

```
PriParallel mainProgram = new PriParallel(
    new CSProcess[] {
        ProcessA,
        ProcessB,
        ProcessC
    }
).run();
```

Em relação a prioridades envolvendo escolhas, tanto internas como externas, foi utilizada a implementação derivada da semântica ALT e PRIALT da linguagem

OCCAM para JCSP e formalizada mais tarde pela semântica definida em $\mathcal{CSPP}$ (LAWRENCE, 2002, 2003a, 2004, 2003b). ALT (ALTernative) e PRIALT (PRIority ALTernative) basicamente são comandos de primitivas de operadores CSP, em que um processo pode escolher entre diversos canais ou eventos, sendo que no caso de PRIALT, a sua escolha segue uma ordem de prioridade pré-definida.

Dados dois canais *a* e *b*, a notação $a \overleftarrow{\Box} b$ pode ser implementada como:

```
final Guard[] guards = {a, b};
final int A = 0;
final int B = 1;
final Alternative alt = new Alternative (guards);
switch (alt.priSelect ()) {
    case A:
        ... // Processo A
        break;
    case B:
        ... // Processo B
        break;
}
```

Caso, caso os dois canais *a* e *b* estejam disponíveis, a prioridade é dada ao canal *a*. A implementação de diversos níveis de ALT e PRIALT entre canais e a sua implementação utilizando kroc (WOOD; WELCH, 1996) e JCSP (P.D.AUSTIN; P.H.WELCH, 2000), de modo a não introduzir problemas de performance podem ser visualizadas em Hilderink e Broenink (2000).

5.2.3 Implementação de Eventos Assíncronos

Pode-se destacar que no caso de eventos assíncronos, a escolha externa envolve uma outra classe disponibilizada pela biblioteca JCSP, sendo *CSTimer* um canal relacionado cuja habilitação é disparada após um período pré determinado, fazendo com que o processo de tempos em tempos verifique se existe algum evento requisitado no canal relacionado a eventos, cujo tratamento é feito baseado em uma fila FIFO (*First In First Out*). A implementação utilizando a classe *CSTimer* pode ser visualizada logo abaixo:

```
CSTimer tim = new CSTimer();
long timeout = tim.read () + interval;
final int EVENTOS_CHANNEL = 0;
final int TIMEOUT_CHANNEL = 1;
tim.setAlarm (timeout);
// Tempo para se efetuar o disparo do canal
final Guard[] guards = {a, b, tim};
```

```
guards[EVENTOS_CHANNEL] = eventosChannel;
guards[TIMEOUT_CHANNEL] = timeoutTimer;
final Alternative alt = new Alternative (guards);
switch (alt.priSelect ()) {
    case EVENTOS_CHANNEL:
        ... // Processo A
        break;
    case TIMEOUT_CHANNEL:
        ... // Processo relacionado ao CSTimer
        break;
}
```

Cada canal utilizado pela simulação é compartilhado por apenas dois processos no máximo, uma limitação em relação à utilização de canais pôde ser observada em relação à impossibilidade de uma pré-verificação dos dados de entrada do canal antes de se efetuar a sincronização. Esta limitação existente já não ocorre na versão Kroc mais recente, chamada de *extended rendevouz* (BARNES; WELCH, 2002a, 2002b), na qual a biblioteca JCSP não implementa.

5.3 Experimentos Realizados

Para validar o funcionamento correto do estudo de caso, realizamos os seguintes testes:

- Teste de verificação do processo de leitura de dados.
- Teste de validação dos eventos assíncronos, ou seja, o funcionamento correto da interface homem-máquina listadas abaixo:
  - Seleção de três níveis de velocidade da mesa.
  - Parada de emergência da mesa.
  - Iniciar a operação da mesa.
- Teste de validação do processo de carregamento de cana-de-açúcar, ou seja, o funcionamento correto das seguintes funcionalidades:
  - Notificação da chegada de caminhão.
  - Reconhecimento correto do tipo de caminhão.
  - Descarregamento correto no fosso por correntes.
  - Descarregamento correto no fosso por tombamento.

- Descarregamento correto na produção por correntes.
- Descarregamento correto na produção por tombamento.
- Funcionamento correto da lavagem nas mesas 1 e 2.
- Funcionamento correto do transporte por garras.

5.3.1 Leitura de dados

Através da utilização de um sistema de arquivamento de resultados, cada um dos aspectos acima foi arquivado em um arquivo distinto para uma melhor análise. Com relação ao processo de leitura de dados, os seguintes resultados foram obtidos:

```
22:52:29,421 DEBUG - DataListeners - Inicio do processo de
                                     leitura ...
22:52:29,562 DEBUG - DataListeners - DataID: 4,
                                     SensorRotacionador: 2
22:52:29,625 DEBUG - DataListeners - DataID: 8,
                                     SensorPesoMesa1: 20
22:52:29,703 DEBUG - DataListeners - DataID: 3,
                                     SensorGancho: 0
22:52:29,765 DEBUG - DataListeners - DataID: 7,
                                     SensorNFeixesCaminhaoData: 22
22:52:29,828 DEBUG - DataListeners - DataID: 2,
                                     SensorTipoCaminhao: 1
22:52:29,906 DEBUG - DataListeners - DataID: 9,
                                     SensorPesoMesa2: 25
22:52:29,984 DEBUG - DataListeners - DataID: 6,
                                     SensorCanaFosso: false
22:52:30,046 DEBUG - DataListeners - DataID: 1,
                                     SensorCaminhao: false
22:52:30,125 DEBUG - DataListeners - DataID: 5,
                                     SensorCabo: 0
22:52:30,125 DEBUG - DataListeners - Fim do processo de
                                     leitura ...

22:52:30,421 DEBUG - DataListeners - Inicio do processo de
                                     leitura ...
22:52:30,484 DEBUG - DataListeners - DataID: 8,
                                     SensorPesoMesa1: 20
22:52:30,546 DEBUG - DataListeners - DataID: 9,
                                     SensorPesoMesa2: 25
22:52:30,546 DEBUG - DataListeners - Fim do processo de
                                     leitura ...

22:52:31,421 DEBUG - DataListeners - Inicio do processo de
                                     leitura ...
22:52:31,515 DEBUG - DataListeners - DataID: 4,
                                     SensorRotacionador: 2
22:52:31,578 DEBUG - DataListeners - DataID: 8,
                                     SensorPesoMesa1: 20
22:52:31,640 DEBUG - DataListeners - DataID: 3,
```

```
                                             SensorGancho: 0
22:52:31,703 DEBUG - DataListeners - DataID: 2,
                                             SensorTipoCaminhao: 1
22:52:31,765 DEBUG - DataListeners - DataID: 9,
                                             SensorPesoMesa2: 25
22:52:31,828 DEBUG - DataListeners - DataID: 6,
                                             SensorCanaFosso: false
22:52:31,890 DEBUG - DataListeners - DataID: 1,
                                             SensorCaminhao: false
22:52:31,953 DEBUG - DataListeners - DataID: 5,
                                             SensorCabo: 0
22:52:31,953 DEBUG - DataListeners - Fim do processo de
                                             leitura ...

22:52:32,421 DEBUG - DataListeners - Inicio do processo de
                                             leitura ...
22:52:32,484 DEBUG - DataListeners - DataID: 8,
                                             SensorPesoMesa1: 20
22:52:32,546 DEBUG - DataListeners - DataID: 7,
                                             SensorNFeixesCaminhaoData: 22
22:52:32,609 DEBUG - DataListeners - DataID: 9,
                                             SensorPesoMesa2: 25
22:52:32,609 DEBUG - DataListeners - Fim do processo de
                                             leitura ...

22:52:33,421 DEBUG - DataListeners - Inicio do processo de
                                             leitura ...
22:52:33,484 DEBUG - DataListeners - DataID: 4,
                                             SensorRotacionador: 2
22:52:33,546 DEBUG - DataListeners - DataID: 8,
                                             SensorPesoMesa1: 20
22:52:33,609 DEBUG - DataListeners - DataID: 3,
                                             SensorGancho: 0
22:52:33,671 DEBUG - DataListeners - DataID: 2,
                                             SensorTipoCaminhao: 1
22:52:33,734 DEBUG - DataListeners - DataID: 9,
                                             SensorPesoMesa2: 25
22:52:33,796 DEBUG - DataListeners - DataID: 6,
                                             SensorCanaFosso: false
22:52:33,859 DEBUG - DataListeners - DataID: 1,
                                             SensorCaminhao: false
22:52:33,921 DEBUG - DataListeners - DataID: 5,
                                             SensorCabo: 0
22:52:33,921 DEBUG - DataListeners - Fim do processo de
                                             leitura ...
```

Pode-se observar que o processo de leitura respeita a taxa de leitura pré-definida, descrita na tabela 5.2, além do fato que a chamada do processo de leitura ocorre a cada 1000 milisegundos, funcionando corretamente, apesar da simulação em si não operar dentro de um sistema operacional de tempo real.

Tabela 5.2: Taxa de leitura dos sensores.

Sensor	Taxa de leitura
SensorCaminhaoData	1 vez a cada 2 ciclos
SensorTipoCaminhaoData	1 vez a cada 2 ciclos
SensorGanchoData	1 vez a cada 2 ciclos
SensorRotacionadorData	1 vez a cada 2 ciclos
SensorCaboData	1 vez a cada 2 ciclos
SensorExisteCanaNoFosso	1 vez a cada 2 ciclos
SensorNFeixesCaminhao	1 vez a cada 4 ciclos
SensorPesoMesa1	1 vez a cada ciclo
SensorPesoMesa2	1 vez a cada ciclo
SensorVolumeCaminhao	1 vez a cada 2 ciclos

5.3.2 Leitura de eventos assíncronos

Com relação ao funcionamento de eventos assíncronos, todos os botões, descritos pela figura 5.5, funcionaram de acordo com o esperado, podendo-se visualizar seus resultados pelos dados de testes descritos abaixo:

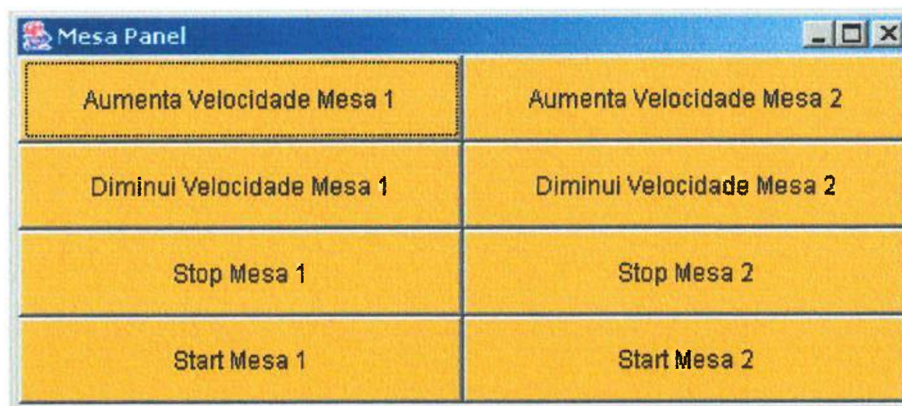


Figura 5.5: Painel de eventos assíncronos.

```
...
23:52:24,296 DEBUG - MesaPesoProcess - Button Aumenta Veloc.
                                       Mesa 1 pressed ...
23:52:25,718 DEBUG - MesaPesoProcess - Button Diminui Veloc.
                                       Mesa 1 pressed ...
23:52:27,796 DEBUG - MesaPesoProcess - Button Stop
                                       Mesa 1 pressed ...
...
```

Além do funcionamento correto dos eventos assíncronos, as condições de contorno especificadas foram atendidas, ou seja, as velocidades das mesas eram aumentadas ou diminuídas de acordo com os requisitos.

5.3.3 Recepção de cana-de-açúcar

Para testar o processo de carregamento de cana-de-açúcar quanto ao funcionamento correto, diversos casos de testes foram realizados, pela alteração dos valores dos sensores através do banco de dados, bem como a utilização do painel de controle de eventos. Em todos os casos, os resultados gravados em arquivo foram condizentes com o esperado.

Além do funcionamento correto do sistema, a especificação do mesmo descrevia situações em que a prioridade de alguma sub-parte sobrepunha em relação à outras, ou seja:

- A mesa número 2 (cana-de-açúcar proveniente de caminhões) tem uma prioridade maior do que a mesa número 1 (cana-de-açúcar proveniente do fosso).
- O descarregamento da cana-de-açúcar na mesa número 2 tem uma prioridade maior do que o descarregamento no fosso.

Testando-se uma situação na qual as duas condições eram habilitadas, o sistema comporta-se de acordo com a especificação adotada.

Os testes realizados tiveram sucesso, validando o módulo de carregamento de cana-de-açúcar.

6 Conclusão

Este trabalho tratou do desenvolvimento de sistema de controle utilizando-se padrões de projeto e linguagens de especificações formais. Quanto aos padrões de projeto utilizados, a abordagem utilizada foi a dos padrões de projeto relacionados ao desenvolvimento de sistemas de controle. A linguagem de especificação formal utilizada por este trabalho foi a combinação das linguagens *Object-Z* e CSP.

A implementação do sistema pode ser dividida em sete etapas.

- A primeira etapa envolve, através dos requisitos do sistema, a definição dos dados de entrada e saída que a compõem. Para cada dado de entrada e saída definido, é associado um identificador único, utilizado por todo o sistema para a leitura e escrita na classe *DataBus*.
- A segunda etapa envolve a definição de cada um dos subprocessos que compõem o sistema, a taxa de atualização e a prioridade de cada um dos processos.
- A terceira etapa envolve a definição da interface com o usuário e a definição dos processos assíncronos.
- A quarta etapa envolve a especificação do sistema através da utilização da linguagem *Object-Z*.
- A quinta etapa envolve modificar as interfaces dos objetos de modo a implementar as sincronizações e comunicações desejadas.
- A sexta etapa envolve a utilização de operadores CSP para a sincronização e comunicação dos objetos.
- E por fim, a sétima etapa envolve a conversão da especificação CSP-OZ em uma linguagem de programação. A linguagem de programação deve ser escolhida de modo que implemente tanto aspectos de orientação a objeto bem como a implementação de operadores CSP.

6.1 Principais contribuições

As principais contribuições deste trabalho podem ser resumidas nos seguintes tópicos:

- Através da utilização de *Object-Z* e CSP pôde-se analisar as principais características que compõem uma arquitetura de sistemas de tempo real, analisando-se o sistema em relação à distribuição de recursos, ao escalonamento de processos, à segurança e ao tratamento de eventos assíncronos. Como base do estudo, foram utilizados os principais padrões de projeto utilizados em sistemas de tempo real comprovando sua aplicabilidade, eficiência e seu auxílio no desenvolvimento de software.

- A utilização de padrões de projeto, melhorou a qualidade do software, em relação à robustez, consistência, reusabilidade, simplicidade e demais propriedades no estudo de caso. A sua utilização como base para uma arquitetura inicial, junto com a utilização de métodos formais, foram as principais contribuições no desenvolvimento e na análise de uma solução aplicada a um estudo de caso. A implementação desta arquitetura inicial diminui o ciclo de desenvolvimento, uma vez que a solução do problema parte de uma arquitetura previamente analisada e correta.

- Pode-se concluir que a utilização de uma especificação formal integrada, descrevendo tanto os aspectos dinâmicos como os aspectos estáticos de um sistema aumentam a compreensibilidade e análise correta de um sistema, retirando eventuais ambigüidades que uma especificação separada poderia introduzir.

- Pode-se concluir através deste trabalho, que a utilização de métodos formais auxilia o desenvolvimento e análise correta do sistema, permitindo a detecção de possíveis erros nas fases iniciais de um projeto.

- Através da utilização de CSP, comprovou-se a eficiência e simplicidade em se construir sistemas concorrentes, sem a preocupação de se utilizar primitivas de sincronização como semáforos, mutex e monitores, facilitando a sua análise e modelagem de maneira correta, simples e transparente.

- Através da utilização de bibliotecas com suporte a CSP como JSCP, comprovou-se a aplicabilidade de métodos formais na especificação, no desenvolvimento e na implementação de sistemas concorrentes de uma maneira transparente.

- A construção de um modelo inicial de um sistema de tempo real, analisado no capítulo 3, podendo servir de ponto de partida inicial para outros sistemas, possibilitando um desenvolvimento mais rápido, eficiente e correto de

eventuais sistemas que tomem como base inicial a arquitetura estudada por este trabalho.

6.2 Trabalhos Futuros

- Analisar melhor as propriedades do sistema, através da utilização de ferramentas de auxílio como provadores automáticos de teoremas como ProofPower (ARTHAN, 2000) para as propriedades estáticas, definidas através de *Object-Z* e checadores de modelo como o FDR (GOLDSMITH, 2001), para as propriedades dinâmicas definidas através da linguagem CSP, para auxiliar em uma modelagem mais correta sistema. A seção 4.3 apresenta propriedades do estudo de caso que poderiam ser verificados através da utilização correta destes aplicativos.

- A construção de aplicativos gráficos de modelagem de sistema de tempo real, com base no estudo efetuado por este trabalho, bem como a construção de *frameworks.* Devido à flexibilidade da solução adotada, muito dos aspectos abordados por este trabalho poderiam ser mecanizados através da construção destes *frameworks* de modo a diminuir o ciclo de desenvolvimento de software.

- Implementação e estudo em sistemas reais, de modo a analisar os impactos existentes entre a mudança de sensores e atuadores simulados para sensores e atuadores reais, bem como o impacto proveniente de mudança de sistema operacional e demais características de hardware existente no sistema real.

- A implementação de um simulador para sistemas de controle distribuídos. Muitas das bibliotecas com suporte a CSP implementam canais CSP com suporte a comunicação distribuída (BROWN, 2004; WELCH; VINTER, 2002), de modo a permitir a implementação de simuladores distribuídos.

- Avaliar o impacto existente na modelagem do sistema quando os processos de leitura e escrita de dados são separados em *n* diferentes processos, com prioridades e freqüência distintas umas das outras.

Referências

ARTHAN, R. *PowerProof reference page.* [S.l.], 2000. Disponível em: http://www.lemma-one.com/ProofPower/index/index.html.

BARNES, F.; WELCH, P. P. H. Prioritised Dynamic Communicating Processes - Part I. In: PASCOE, J.; LOADER, R.; SUNDERAM, V. (Ed.). *Communicating Process Architectures 2002.* [S.l.: s.n.], 2002. p. 321–352. ISBN 1 58603 268 2.

BARNES, F.; WELCH, P. P. H. Prioritised Dynamic Communicating Processes - Part II. In: PASCOE, J.; LOADER, R.; SUNDERAM, V. (Ed.). *Communicating Process Architectures 2002.* [S.l.: s.n.], 2002. p. 353–370. ISBN 1 58603 268 2.

BATE, I.; CERVIN, A.; NIGHTINGALE, P. Establishing timing requirements and control attributes for control loops in real-time systems. In: *Proceedings of the 15th Euromicro Conference on Real-Time Systems.* [S.l.: s.n.], 2003. p. 121–128.

BOLLELLA, G. *The Real Time Specification for Java - The Real-Time for Java Expert Group.* [S.l.], 2000. Disponível em: http://www.rtj.org/.

BOWEN, J. P. Formal methods in safety-critical standards. In: *Proc. 1993 Software Engineering Standards Symposium (SESS'93), Brighton, UK.* IEEE Computer Society Press, 1993. p. 168–177. Disponível em: <citeseer.ist.psu.edu/article/bowen93formal.html>.

BOWEN, J. P.; HINCHEY, M. G. Seven more myths of formal methods. *IEEE Software*, v. 12, n. 3, p. 34–41, 1995. Disponível em: <citeseer.ist.psu.edu/bowen95seven.html>.

BOWEN, J. P.; HINCHEY, M. G. Ten commandments of formal methods. *IEEE Computer*, v. 28, n. 4, p. 56–63, 1995. Disponível em: <citeseer.ist.psu.edu/bowen95ten.html>.

BOWEN, J. P.; HINCHEY, M. G. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In: *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems.* New York, NY, USA: ACM Press, 2005. p. 8–16. ISBN 1-59593-148-1.

BOWEN, M. G. H. J. P. Seven more myths of formal methods: Dispelling industrial prejudices. In: BERTRAN, M.; NAFTALIN, M.; DENVIR, T. (Ed.). *FME'94: Industrial Benefit of Formal Methods.* Springer-Verlag, 1994. (Lecture Notes in Computer Science, v. 873), p. 105–117. Disponível em: <citeseer.ist.psu.edu/article/bowen94seven.html>.

BROWN, N.; WELCH, P. An Introduction to the Kent C++CSP Library. In: BROENINK, J.; HILDERINK, G. (Ed.). *Communicating Process Architectures 2003.* [S.l.: s.n.], 2003. p. 139–156. ISBN 1586033816.

BROWN, N. C. C++CSP Networked. In: EAST, D. I. R.; DUCE, P. D.; GREEN, D. M.; MARTIN, J. M. R.; WELCH, P. P. H. (Ed.). *Communicating Process Architectures 2004.* [S.l.: s.n.], 2004. p. 185–200. ISBN 1 58603 458 8.

CAVALCANTI, A. L. C.; SAMPAIO, A. C. A. From CSP-OZ to Java with Processes. *In IPDPS (International Parallel and Distributed processing Symposium) 2002 Workshop on Formal Methods for Parallel Programming*, 2002.

DAHL, O.-J.; DIJKSTRA, E.; HOARE, C. *Structured Programming.* [S.l.]: Academic Press, 1972. (A.P.I.C. Studies in Data Processing, v. 8).

DERRICK, J.; SMITH, G. Structural refinement of systems specified in Object-Z and CSP. *Formal Aspects of Computing*, v. 15, p. 1–27, 2003.

DOUGLASS, B. P. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems.* [S.l.]: Addison Wesley, 2002. 528 p. ISBN 0-201-69956-7.

FISCHER, C. CSP-OZ: A combination of Object-Z and CSP. *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, v. 2, p. 423–438, 1997.

FISCHER, C. *Combination and Implementation of Processes and Data: from CSP-OZ to Java.* Tese (Doutorado) — University of Oldenburg, 2000.

FISCHER, C.; WEHRHEIM, H. Behavioural Subtyping Relations for Object-Oriented Formalisms. *Algebraic Methodology and Software Technology*, v. 1816, p. 469–160, 2000.

FISCHER, C.; WEHRHEIM, H. Failure-Divergence Semantics as a Formal Basis for an Object-Oriented Integrated Formal Method. *Current Trends in Theoretical Computer Science: The Challenge of the New Century*, v. 2, 2004.

GAMMA, E.; HELM, R.; JOHNSON, R. *Design Patterns. Elements of Reusable Object-Oriented Software.* [S.l.]: Addison-Wesley, 1995. 395 p. (Addison-Wesley Professional Computing Series). GAM e 95:1 1.Ex. ISBN 0201633612.

GOLDSMITH, M. *FDR: User Manual and Tutorial, version 2.77.* [S.l.], 2001. Formal Systems (Europe) Ltda.

HALL, A. Seven myths of formal methods. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 7, n. 5, p. 11–19, 1990. ISSN 0740-7459.

HILDERINK, G. H. *CTJ: Communicating Threads for Java.* [S.l.], 2000. Disponível em: http://www.ce.utwente.nl/javapp.

HILDERINK, G. H.; BROENINK, J. F. Conditional Communication in the Presence of Priority. In: WELCH, P. P. H.; BAKKERS, A. W. P. (Ed.). *Communicating Process Architectures 2000.* [S.l.: s.n.], 2000. p. 77–98. ISBN 1 58603 077 9.

HOARE, C. A. R. Communicating Sequential Processes. *Communications of the ACM*, v. 21, p. 666–677, 1978.

HOARE, C. A. R. *Communicating Sequential Processes.* [S.l.]: Prentice Hall International, 1985.

HOARE, C. A. R. (Ed.). *Occam 2.1 Reference Manual: Inmos Limited.* [S.l.: s.n.], 1988. Disponível em: http://www.wotug.org/occam.

HOARE, C. A. R. The verifying compiler: A grand challenge for computing research. *J. ACM*, ACM Press, New York, NY, USA, v. 50, n. 1, p. 63–69, 2003. ISSN 0004-5411.

JOSEPH, M. (Ed.). *Real Time Systems Specification Verification and Analysis.* [S.l.]: Prentice Hall, 2001. 272 p. ISBN 0-13-455297-0.

KASSEL, G.; SMITH, G. Model checking object-z classes: Some experiments with fdr. In: *APSEC '01: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2001. p. 445.

KELLY, T.; MCDERMID, J. Software in safety critical systems: Achievement and prediction. In: *Proceedings of the 5th International Conference on Control and Instrumentation in Nuclear Installations*. [S.l.: s.n.], 2004.

LAWRENCE, D. A. E. Acceptances, Behaviours and infinite activity in CSPP. *In Proceedings of Communicating Process Architectures*, IOS Press, p. 17–38, 2002.

LAWRENCE, D. A. E. Overtures and hesitant offers: hiding in CSPP. *Proceedings of Communicating Process Architectures*, p. 97–109, 2003.

LAWRENCE, D. A. E. The theory of CSPP. 2003.

LAWRENCE, D. A. E. Observing Processes. In: EAST, D. I. R.; DUCE, P. D.; GREEN, D. M.; MARTIN, J. M. R.; WELCH, P. P. H. (Ed.). *Communicating Process Architectures 2004*. [S.l.: s.n.], 2004. p. 147–156. ISBN 1 58603 458 8.

MAHONY, B.; DONG, J. Active Objects in TCOZ. *2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, IEEE Computer Society Press, 1998.

MAHONY, B.; DONG, J. Sensors and Actuators in TCOZ. *Lecture Notes in Computer Science*, v. 1709, 1999.

MAHONY, B.; DONG, J. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, v. 26, p. 150–177, 2000.

MAHONY, B.; DONG, J. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing journal*, v. 13, p. 14–160, 2002.

MARTIN, J. *The Design and Construction of Deadlock-Free Concurrent Systems*. Tese (Doutorado) — University of Buckingham, 1996.

MILNER, R. *Communication and Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989. ISBN 0131149849.

MILNER, R. The polyadic π-calculus: A tutorial. In: . [S.l.: s.n.], 1991. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

MILNER, R. *Communicating and mobile systems: the π-calculus*. New York, NY, USA: Cambridge University Press, 1999. ISBN 0-521-65869-1.

MOTA, A.; BORBA, P.; SAMPAIO, A. *Mechanical abstraction of CSPZ processes*. 2002. Disponível em: <citeseer.ist.psu.edu/mota02mechanical.html>.

MOTA, A.; SAMPAIO, A. Model-checking CSP-Z. *Lecture Notes in Computer Science*, v. 1382, p. 205–220, 1998. Disponível em: <citeseer.ist.psu.edu/mota98modelchecking.html>.

MOTA, A.; SAMPAIO, A. Model-Checking CSP-Z: Strategy, Tool Support and Industrial Application. *Sci. Comput. Program.*, v. 40, p. 59–96, 2001. Disponível em: <citeseer.ist.psu.edu/387905.html>.

NEVISON, C. Teaching Distributed and Parallel Computing with Java and CSP. In: *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. [S.l.]: IEEE Computer Society, 2001. p. 484. ISBN 0-7695-1010-8.

P.D.AUSTIN; P.H.WELCH. *Java Communicating Sequential Processes*. [S.l.], 2000. Disponível em: http://www.cs.ukc.ac.uk/projects/ofa/jcsp.

P.H.WELCH; J.M.R.MARTIN. A CSP Model for Java Multithreading. In: P.NIXON; I.RITCHIE (Ed.). *Software Engineering for Parallel and Distributed Systems.* IEEE Computer Society Press, 2000. p. 114–122. ISBN 0-7695-0634-8. Disponível em: <http://www.cs.ukc.ac.uk/pubs/2000/1150>.

RAJU, V.; RONG, L.; STILES, G. S. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In: BROENINK, J. F.; HILDERINK, G. H. (Ed.). *Communicating Process Architectures 2003.* [S.l.: s.n.], 2003. p. 63–81. ISBN 1 58603 381 6.

ROSCOE, A. W. *The Theory and Practice of Concurrency.* [S.l.]: Prentice Hall, 1998.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *Unified Modeling Language Reference Manual, The (2nd Edition).* [S.l.]: Addison-Wesley Professional, 1998. 576 p. ISBN 020130998X.

SCHALLER, N. C.; HILDERINK, G. H.; WELCH, P. P. H. Using Java for Parallel Computing - JCSP versus CTJ. In: WELCH, P. P. H.; BAKKERS, A. W. P. (Ed.). *Communicating Process Architectures 2000.* [S.l.: s.n.], 2000. p. 205–226. ISBN 1 58603 077 9.

SCHNEIDER, S. *Concurrent and Real-time Systems.* [S.l.]: John Wiley and Sons, Ltd., 2000.

SHALLOWAY, J. T. A. *Design Patterns Explained : A New Perspective on Object-Oriented Design (2nd Edition) (Software Patterns Series).* [S.l.]: Addison-Wesley Professional, 2004. 480 p. ISBN 0321247140.

SMITH, G. A fully abstract semantics of classes for object-Z. *Formal Aspects of Computing*, v. 7, p. 30–65, 1995.

SMITH, G. *The Object-Z specification language.* [S.l.]: Kluwer Academic Publisher, 2000.

SMITH, G. *Recursive Schema Definitions in Object-Z.* [S.l.]: Springer-Verlag, 2000. 42–58 p.

SMITH, G.; DERRICK, J. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in System Design*, v. 18, p. 249–284, 2001.

SMITH, G.; HAYES, I. Structuring Real-Time Object-Z Specifications. *Lecture Notes in Computer Science*, v. 1945, 2000.

SMITH, G.; HAYES, I. An introduction to Real-Time Object-Z. *Formal Aspects of Computing*, v. 13, p. 128–141, 2002.

SMITH, R. D. G. R. e G. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, v. 17, p. 511–533, 1995.

SPIVEY, J. M. *The Z notation: A reference manual.* Prentice Hall, 1992. Disponível em: <http://spivey.oriel.ox.ac.uk/ mike/zrm/.>.

TORNGREN, M. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Syst.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 14, n. 3, p. 219–250, 1998. ISSN 0922-6443.

VILLANI, E. *Modelagem e Análise de Sistemas Supervisórios Híbridos.* Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo, 2003.

WEHRHEIM, H. Behavioural subtyping and property preservation. *FMOODS'00: Formal Methods for Open Object-Based Distributed Systems*, 2000.

WEHRHEIM, H. Patterns and Rules for Behavioural Subtyping. *FORTE*, p. 335–352, 2001.

WELCH, P. H.; VINTER, B. Cluster Computing and JCSP Networking. In: PASCOE, J.; LOADER, R.; SUNDERAM, V. (Ed.). *Communicating Process Architectures 2002*. [S.l.: s.n.], 2002. p. 203–222. ISBN 1 58603 268 2.

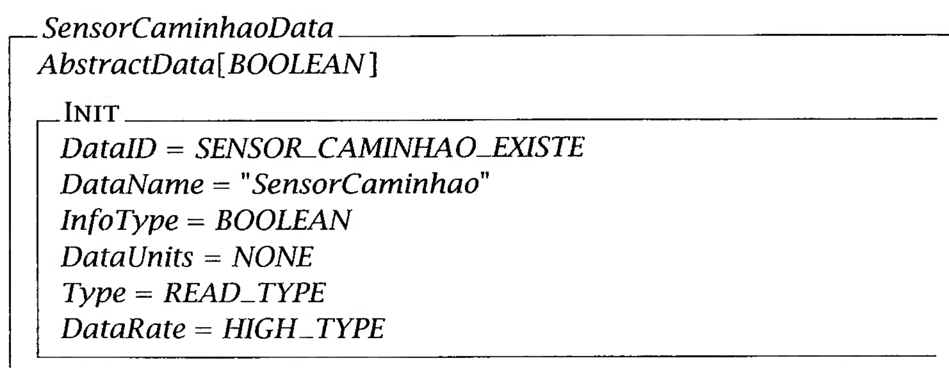
WILSON, S.; KELLY, T.; MCDERMID, J. *Safety Case Development: Current Practice, Future Prospects*. 1997. Disponível em: <citeseer.ist.psu.edu/wilson97safety.html>.

WOOD, D. C.; WELCH, P. H. The kent retargetable occam compiler. In: *WoTUG '96: Proceedings of the 19th world occam and transputer user group technical meeting on Parallel processing developments*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 1996. p. 143–166. ISBN 90-5199-261-0.

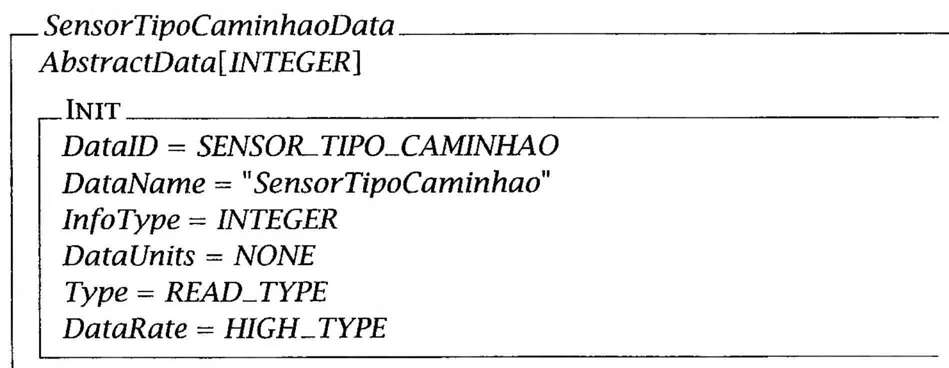
Apêndice A - Diagramas do estudo de caso

A.1 Diagramas de sensores

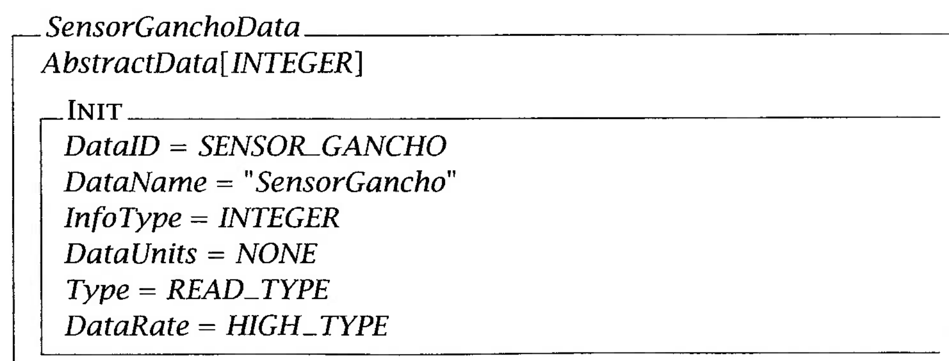
Sensor de existência de caminhão.


SensorCaminhaoData
AbstractData[*BOOLEAN*]
INIT
DataID = *SENSOR_CAMINHAO_EXISTE*
DataName = "*SensorCaminhao*"
InfoType = *BOOLEAN*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

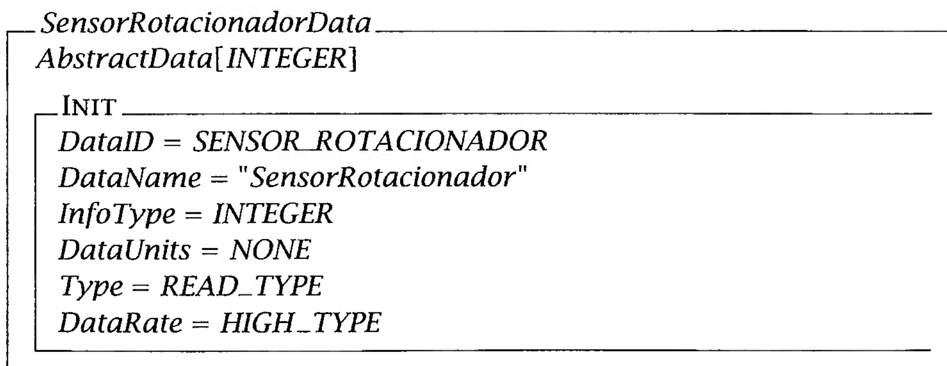
Sensor do tipo de caminhão.


SensorTipoCaminhaoData
AbstractData[*INTEGER*]
INIT
DataID = *SENSOR_TIPO_CAMINHAO*
DataName = "*SensorTipoCaminhao*"
InfoType = *INTEGER*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

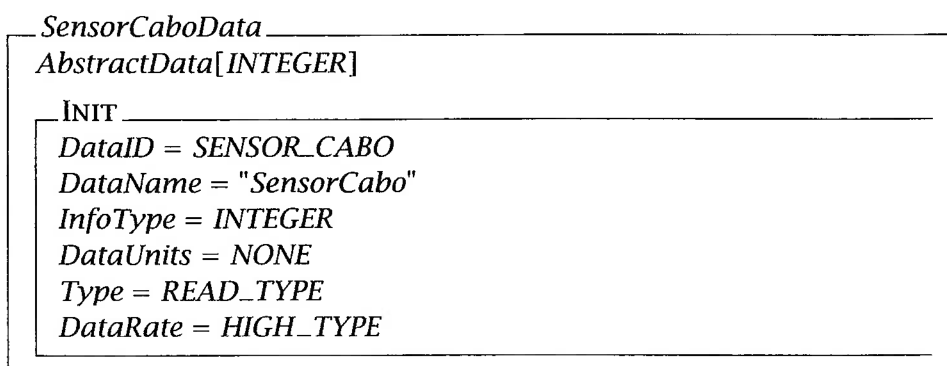
Sensor do estado de posicionamento do gancho.


SensorGanchoData
AbstractData[*INTEGER*]
INIT
DataID = *SENSOR_GANCHO*
DataName = "*SensorGancho*"
InfoType = *INTEGER*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

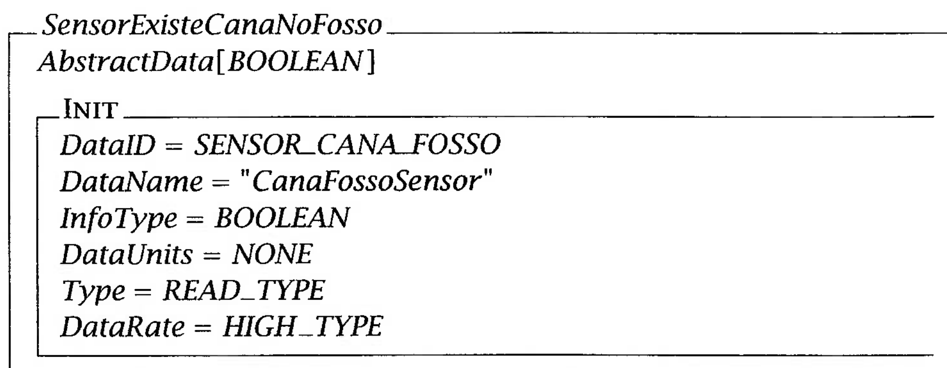
Sensor responsável pela leitura do rotacionador do guindaste.


SensorRotacionadorData
AbstractData[*INTEGER*]
INIT
DataID = *SENSOR_ROTACIONADOR*
DataName = "*SensorRotacionador*"
InfoType = *INTEGER*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

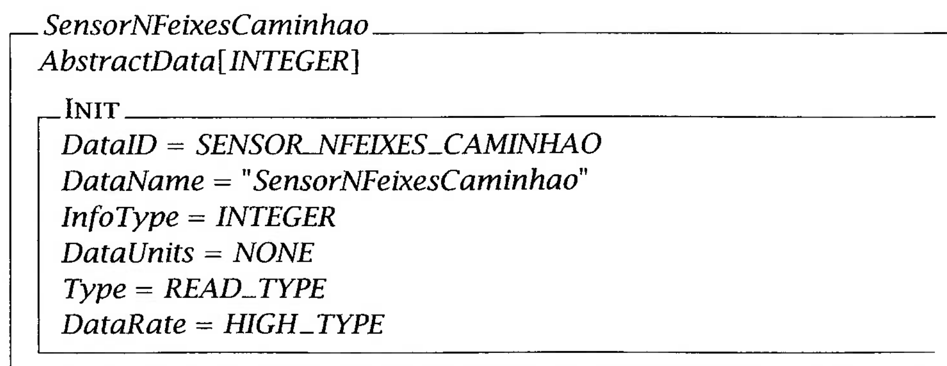
Sensor responsável pela leitura do cabo do guindaste.


SensorCaboData
AbstractData[*INTEGER*]
INIT
DataID = *SENSOR_CABO*
DataName = "*SensorCabo*"
InfoType = *INTEGER*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

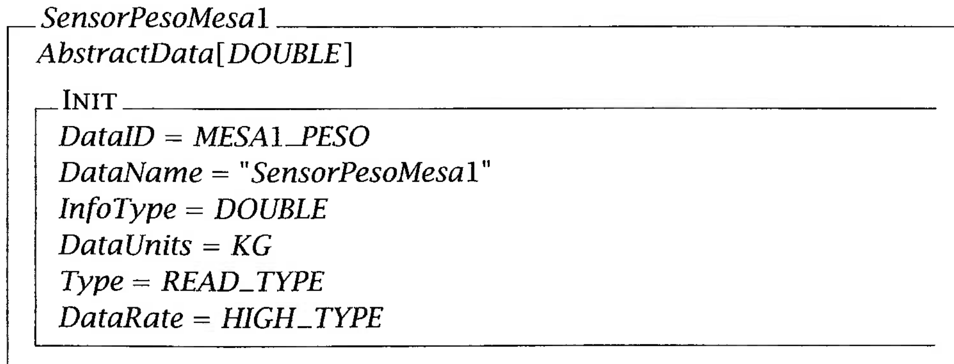
Sensor responsável pela leitura da existência de cana no fosso.


SensorExisteCanaNoFosso
AbstractData[*BOOLEAN*]
INIT
DataID = *SENSOR_CANA_FOSSO*
DataName = "*CanaFossoSensor*"
InfoType = *BOOLEAN*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

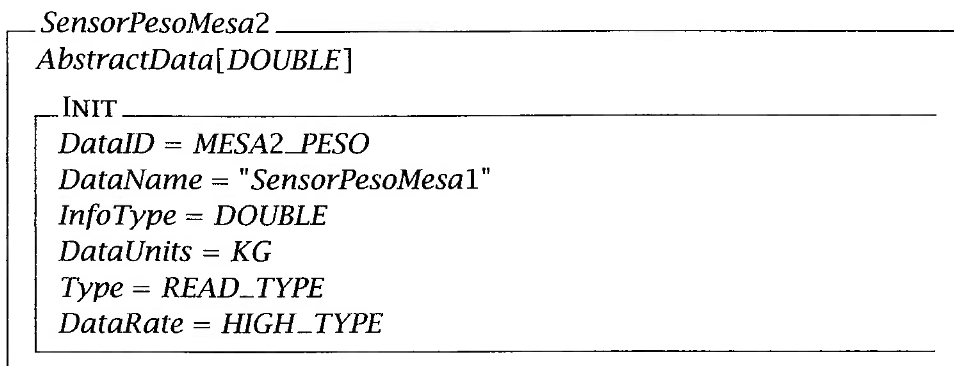
Sensor responsável pela leitura da quantidade de feixes de cana de açúcar.


SensorNFeixesCaminhao
AbstractData[*INTEGER*]
INIT
DataID = *SENSOR_NFEIXES_CAMINHAO*
DataName = "*SensorNFeixesCaminhao*"
InfoType = *INTEGER*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

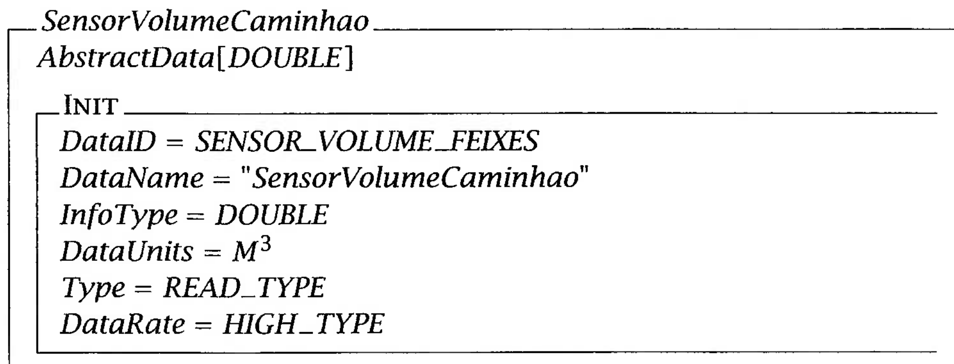
Sensor responsável pela leitura do peso da mesa 1.



Sensor responsável pela leitura do peso da mesa 2.



Sensor responsável pela leitura do volume do caminhão.



A.2 Diagramas de atuadores

Atuador responsável pelo gancho do guindaste.

```
AtuadorGanchoData
AbstractData[BOOLEAN]
  INIT
  DataID = ATUADOR_GANCHO
  DataName = "AtuadorGancho"
  InfoType = BOOLEAN
  DataUnits = NONE
  Type = READ_TYPE
  DataRate = HIGH_TYPE
```

Atuador responsável pelo rotacionador do caminhão.

```
AtuadorRotacionadorData
AbstractData[BOOLEAN]
  INIT
  DataID = ATUADOR_ROTACIONDOR
  DataName = "AtuadorRotacionador"
  InfoType = BOOLEAN
  DataUnits = NONE
  Type = READ_TYPE
  DataRate = HIGH_TYPE
```

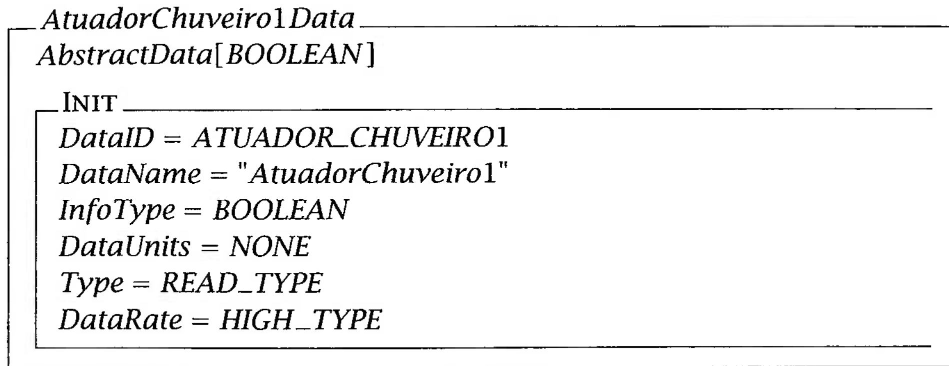
Atuador responsável pelo cabo do guindaste.

```
AtuadorCaboData
AbstractData[BOOLEAN]
  INIT
  DataID = ATUADOR_CABO
  DataName = "AtuadorCabo"
  InfoType = BOOLEAN
  DataUnits = NONE
  Type = READ_TYPE
  DataRate = HIGH_TYPE
```

Atuador responsável pela caçamba do caminhão.

```
AtuadorCacambaData
AbstractData[BOOLEAN]
  INIT
  DataID = ATUADOR_CACAMBA
  DataName = "AtuadorCacamba"
  InfoType = BOOLEAN
  DataUnits = NONE
  Type = READ_TYPE
  DataRate = HIGH_TYPE
```

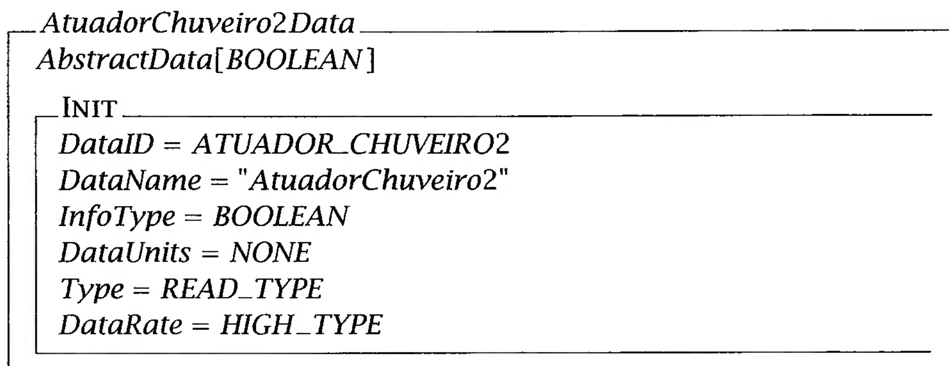
Atuador responsável pelo chuveiro da mesa 1.



AtuadorChuveiro1Data
AbstractData[*BOOLEAN*]

INIT
DataID = *ATUADOR_CHUVEIRO1*
DataName = "*AtuadorChuveiro1*"
InfoType = *BOOLEAN*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

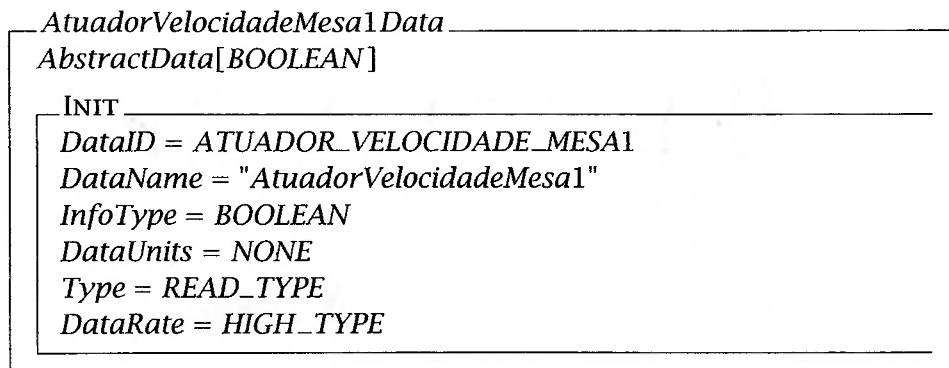
Atuador responsável pelo chuveiro da mesa 2.



AtuadorChuveiro2Data
AbstractData[*BOOLEAN*]

INIT
DataID = *ATUADOR_CHUVEIRO2*
DataName = "*AtuadorChuveiro2*"
InfoType = *BOOLEAN*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

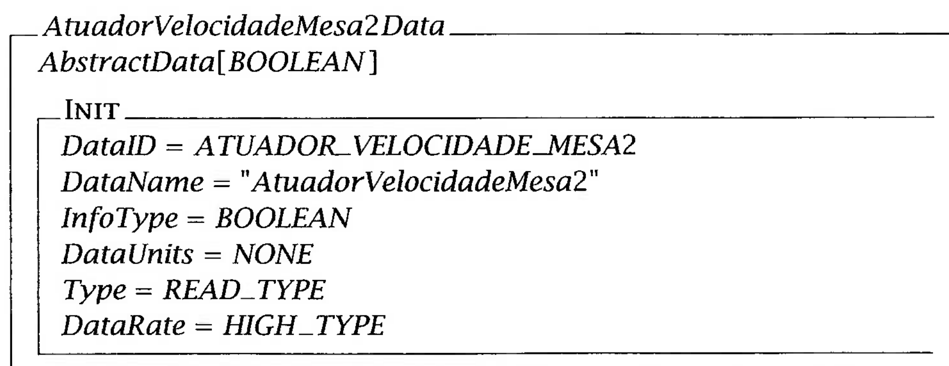
Atuador responsável pela velocidade da mesa 1.



AtuadorVelocidadeMesa1Data
AbstractData[*BOOLEAN*]

INIT
DataID = *ATUADOR_VELOCIDADE_MESA1*
DataName = "*AtuadorVelocidadeMesa1*"
InfoType = *BOOLEAN*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*

Atuador responsável pela velocidade da mesa 2.



AtuadorVelocidadeMesa2Data
AbstractData[*BOOLEAN*]

INIT
DataID = *ATUADOR_VELOCIDADE_MESA2*
DataName = "*AtuadorVelocidadeMesa2*"
InfoType = *BOOLEAN*
DataUnits = *NONE*
Type = *READ_TYPE*
DataRate = *HIGH_TYPE*