PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA BIBLIOTECA DE PADRÕES DE ESPECIFICAÇÃO EM EVENT-B PARA MECANISMOS DE TROCA DE MENSAGENS EM SISTEMA DISTRIBUÍDOS

PAULO JUNIOR PENNA PIVETTA

Dissertação de Mestrado apresentada como requisito para obtenção do título de Mestre em Ciência da Computação pelo Programa de Pós-graduação da Faculdade de Informática. Área de concentração: Ciência da Computação.

Orientador: Prof. Fernando Luís Dotti
Co-orientador: Prof. Avelino Francisco Zorzo

Porto Alegre, Brasil
2012

P693b Pivetta, Paulo Junior Penna
Uma biblioteca de padrões de especificação em Event-B para mecanismos de troca de mensagens em sistema distribuídos / Paulo Junior Penna Pivetta. – Porto Alegre, 2012.
112 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Fernando Luís Dotti.
Co-orientador: Prof. Dr. Avelino Francisco Zorzo.

1. Informática. 2. Sistemas Distribuídos. 2. Protocolos de Comunicação. I. Dotti, Fernando Luís. II. Zorzo, Avelino Francisco. III. Título.

CDD 004.36

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

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

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "Uma Biblioteca de Padrões de Especificação em Event-B para Mecanismos de Troca de Mensagens em Sistema Distribuídos", apresentada por Paulo Junior Penna Pivetta, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 19/03/10 pela Comissão Examinadora:

Prof. Dr. Fernando Luís Dotti – Orientador — PPGCC/PUCRS

Prof. Dr. Paulo Henrique Lemelle Fernandes – PPGCC/PUCRS

Prof. Dr. Álvaro Freitas Moreira – UFRGS

Homologada em...13/..11.../...2012.., conforme Ata No. ...024... pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

AGRADECIMENTOS

Agradeço primeiramente a Deus e à mãe rainha por dar força e esperança nos momentos de fraqueza, iluminando meu dia a dia. Minha fé ajudou superar as barreiras que surgiram e tornou possível a realização desta conquista.

Ao meu pai, mãe e irmão por me ajudarem nos momentos de dificuldades e darem condições para realizar esta conquista, assim como forças para sair de casa e correr atrás dos meus sonhos. Obrigado por acreditarem em mim.

À minha tia Maria Genoveva Penna, meu tio Wilson de Oliveira Miranda e minha prima Mariela Francisca Penna Miranda que me ofereceram um teto para me abrigar e um ambiente familiar para viver todo esse tempo. Minha segunda família.

À minha namorada Sabrina Corrêa que sempre me deu forças e esteve ao meu lado nos momentos bons e ruins. Agradeço por você entender os momentos de ausência e ser minha grande companheira nesta caminhada. Obrigado por ser uma namorada tão dedicada.

Ao meu orientador, Prof. Fernando Luís Dotti que foi importante para que este trabalho se tornasse realidade, tanto pelas considerações como pela amizade. Um orientador dedicado, interessado e sempre presente. Um exemplo a ser seguido.

Ao meu co-orientador, Prof. Avelino Francisco Zorzo pelas contribuições realizadas neste trabalho. Agradeço por ter me acolhido, disponibilizando seu tempo e conhecimento, tornando possível a realização deste trabalho.

Ao grande amigo e colega Dione Taschetto por estar presente em mais um passo de minha vida. Obrigado pela amizade.

Aos demais colegas, professores e funcionários do programa de pós-graduação, obrigado pela atenção, paciência e pelas sugestões.

Ao CNPQ, por dar o apoio financeiro necessário para que eu pudesse realizar esta conquista e à PUCRS pela oportunidade.

Muito obrigado a todos!

UMA BIBLIOTECA DE PADRÕES DE ESPECIFICAÇÃO EM EVENT-B PARA MECANISMOS DE TROCA DE MENSAGENS EM SISTEMA DISTRIBUÍDOS

RESUMO

O desenvolvimento de sistemas distribuídos e protocolos de comunicação é uma tarefa complexa e o uso de técnicas de especificação e verificação formal torna-se necessário para garantir a corretude de tais sistemas.

Enquanto técnicas de model-checking passam pelo problema da explosão do espaço de estados, o uso de provadores de teoremas representa um importante recurso para verificação de sistemas com ilimitado número estados. O método formal Event-B, de uso crescente na indústria e academia, se apóia na técnica de prova de teoremas e suporta refinamento.

A contribuição deste trabalho está em proporcionar uma biblioteca reusável de padrões de especificação, em Event-B, de mecanismos de troca de mensagens em sistemas distribuídos. Um padrão de especificação define a semântica de comunicação desejada em um canal, demostrando formalmente suas propriedades.

Durante o desenvolvimento de um sistema distribuído, o desenvolvedor pode fazer uso destes padrões através de passos guiados de refinamento do sistema. O sistema resultante garante a semantica de comunicação definida no padrão utilizado e livra o desenvolvedor de se preocupar em definir o sistema de comunicação a partir do início e provar suas propriedades.

Palavras-chave: Sistemas Distribuídos; Formalismos; Event-B.

A LIBRARY OF EVENT-B SPECIFICATION PATTERNS FOR MESSAGE PASSING MECHANISMS IN DISTRIBUTED SYSTEM

ABSTRACT

The development of distributed systems and communication protocols is not a trivial task and the use of formal specification and verification techniques becomes necessary to assure the correctness of such systems.

While model-checking techniques face the state space explosion problem, the use of theorem provers is an important resource for verification of systems with unlimited number of states. The formal method Event-B, increasingly being used in both industry and academia, is based on the technique of theorem proving and also supports refinement.

The contribution of this work is a library of reusable formal specification patterns, in Event-B, for message passing mechanisms commonly employed in distributed systems. A specification pattern defines the desired communication semantics of a channel, having its properties formally proven.

During the development of a distributed system, the developer may use these patterns by applying guided refinement steps on the target model. The resulting system is assured to have the communication semantics as defined by the pattern, thus freeing the developer of defining the communication system from scratch and of proving its properties.

Keywords: Distributed System; Formalisms; Event-B.

Lista de Figuras

Lista de Tabelas

SUMÁRIO

1. INTRODUÇÃO

Com o desenvolvimento de microprocessadores poderosos a preços atrativos e o surgimento de redes de computadores de alta velocidade [Tv02], tornou-se possível transferir grandes volume de dados entre máquinas em poucos microsegundos, tornando-se prático compor um sistema utilizando um grande número de computadores conectados. Sistema que chamamos de sistemas distribuídos [Tv02].

A definição de um sistema distribuído difere entre autores, por exemplo, "um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens" [DKC05], ou "um sistema distribuído é uma coleção de computadores independentes que na visão do usuário aparenta ser um único computador" [Tv02].

O desenvolvimento de sistemas distribuídos é uma tarefa complexa, pois exige do desenvolvedor um conhecimento detalhado sobre aspectos do ambiente de execução, que inerentemente é concorrente e sem estado global. Além disso, a construção de sistemas distribuídos empregam dificuldades como:

- Heterogeneidade, que consiste em sistemas desenvolvidos em um ambiente com diversas diferenças (sistemas operacionais diferentes, hardwares diferentes, redes diferentes) prejudicando a interação das diferentes partes. Para contornar este problema, em redes heterogêneas utiliza-se de protocolos de comunicação, mascarando-se as diferenças. Para o restante das diversidades normalmente é utilizado um *middleware* [Tv02], resultando em uma padronização dos componentes do sistema.

- A construção de sistemas distribuídos abertos, ou seja, sistemas onde possam ser adicionados ou substituídos novos componentes. O fato de um sistema distribuído ser ou não um sistema aberto é determinado principalmente pelo grau com que novos serviços podem ser adicionados e disponibilizados para uso por uma variedade de programas clientes. Em um sistema distribuído aberto o maior desafio está em integrar os componentes de diferentes desenvolvedores [DKC05].

- Um sistema distribuído deve manter-se funcional e eficiente a medida que aumenta-se o número de usuários acessando compartilhadamente determinado recursos. Portanto o sistema deve manter a qualidade do serviço prestado, evitando-se possíveis gargalos. A escalabilidade, como é chamada esta propriedade, é um dos problemas centrais encontrados em sistema distribuídos [DKC05].

- Um sistema distruído deve manter-se seguro, ou seja, os dados do sistema que trafegam nas redes podem conter informações sigilosas dos usuários, por exemplo, sistema de serviços bancários, portanto necessitando estar seguros. A segurança da informação é baseada em

três princípios: confidencialidade (pessoas não autorizadas não podem acessar a informação), integridade (proteção dos dados ou informações contra alterações ou danos) e disponibilidade (a informação deve estar disponível quando necessária) [TK99].

- Em um sistema distribuído podem ocorrer falhas de processo independentes, cabendo ao desenvolvedor do sistema o tratamento dessas falhas. O tratamento de falhas torna-se difícil em sistemas distribuídos, pelo fato que alguns componentes podem falhar enquanto outros continuam a executar. Entretanto existem algumas técnicas para tratamento de falhas como: detecção de falhas, mascaramento de falhas, tolerância a falhas, recuperação de falhas e redundância [Tv02].

- Os desenvolvedores de um sistema distribuído devem preocupar-se em torna o sistema transparente para o usuário final, ou seja, as diferentes partes do sistema devem ser vista como um todo.

Dadas as dificuldades para o desenvolvimento de sistemas distribuídos, este trabalho tem como finalidade reduzir os erros de projeto, ocasionados pela não conservação de uma propriedade do sistema. Portanto pretende-se, através da especificação e análise de um sistema distribuído, verificar a corretude dos modelos em fases iniciais de desenvolvimento.

1.1 Motivação

No desenvolvimento de sistemas distribuídos exige-se do desenvolvedor, um conhecimento detalhado sobre os aspectos do sistema empregado e do ambiente. Portanto, torna-se interessante a utilização métodos formais que comprovem que o sistema se comporta de maneira correta, mesmo na existência de um ambiente hostil (existência de falhas), tendo como necessidade o quanto antes possível a afirmação das propriedades funcionais do sistema.

A modelagem é uma técnica de engenharia aprovada e bem aceita no desenvolvimento de sistemas. Quanto maior for a complexidade de um sistema maior será a probabilidade de erro [FS00]. Através de modelos matemáticos é possível analisar os efeitos de um sistema.

Em um modelo é possível verificar as características do sistema e facilitar a comunicação das idéias entre diferentes desenvolvedores, assim como limitar o problema restringindo o mesmo para um único foco. Desse modo pode-se dividir um problema grande em diversos problemas menores.

O uso de métodos formais para especificar algoritmos distribuídos e protocolos de comunicação é recorrente na literatura. Tipicamente, são empregados formalismos baseados em álgebra de processos [Mil99], linguagens de entrada para ambientes tradicionais de verificação, como por exemplo PROMELA/SPIN [Hol97], autômatos de entrada e saída (I/OAutomata [LT89]), formalismos baseados em lógica temporal, como *Temporal Logic of Actions* (TLA) [ZRR02].

Vários de tais métodos dispõem de abstrações propícias à área do problema e tipicamente recaem sobre ferramentas de análise utilizando a estratégia de *model checking*. Esta última é menos complexa para o usuário, se comparada com prova de teoremas, mas os modelos são limitados pelo

problema da explosão do espaço de estados. Outra questão importante, com relação a vários de tais métodos, é a carência de abordagens para gerar implementações comprovadamente corretas a partir das especificações. O uso de técnicas de refinamento é importante neste sentido.

O uso de provadores de teoremas, ainda que uma tarefa não trivial, tem sido facilitado devido ao surgimento de linguagens de especificação e plataformas de suporte voltadas para este fim, como por exemplo Event-B [AM05] e a plataforma RODIN (vide Seção 2.3). É crescente a aplicação de Event-B para modelagem de sistemas distribuídos (vide Seção 2.4).

A utilização do formalismo Event-B na modelagem de sistemas distribuídos, implica em um raciocínio antecipado sobre o problema a ser modelado, como também a garantia que as propriedades do sistema estarão presentes. Com as propriedades desejadas do sistema preservadas diminui-se a possibilidade de erro, ou seja, a chance de ocorrer um erro pela não conservação de uma propriedade na validação do sistema (fase de testes) é menor.

O formalismo aplicado a sistemas complexos resulta em modelos grandes e complexos, tornando-se uma tarefa árdua a sua utilização [YA07]. Entretanto Event-B possibilita o uso de refinamento, que permite especificar modelos em diferentes níveis de abstrações facilitando na modelagem e na prova das propriedades. Além disso, no formalismo Event-B, existe o conceito de padrão que possibilita o reuso de modelos provados formalmente em novos modelos.

1.2 Objetivos

Na linguagem Event-B não existe suporte para representação de mensagens ou canais. A linguagem é basicamente constituída de eventos e variáveis que representam transições de estados em um modelo, portanto dificultando a representação das características de um sistema distribuído na modelagem.

A proposta deste trabalho é a criação de uma biblioteca de padrões de especificação formal em Event-B. Esta biblioteca suporta a representação de diferentes semânticas de comunicação clássicas da literatura, contendo suas propriedades formalmente provadas. Desta forma, liberando o desenvolvedor que estiver modelando um sistema distribuído em Event-B das preocupações quanto a validação da comunicação do seu sistema.

Por fim será desenvolvida uma aplicação distribuída na plataforma RODIN contendo especificações da biblioteca, de forma a demonstrar a aplicação dos padrões em sistemas modelados em Event-B.

1.3 Organização do Trabalho

O restante do trabalho está organizado da seguinte forma: no Capítulo 2 será apresentado o referencial teórico, contendo conceitos sobre o formalismo Event-B e trabalhos relacionados. No Capítulo 3, encontra-se a biblioteca com padrões de comunicação para Event-B. Já no Capítulo 4 é apresentado um estudo de caso, aplicando-se alguns padrões de comunicação, contidos na biblioteca

de padrões. As conclusões são apresentadas no Capítulo 5. Por fim no apêndice A são apresentados sistemas distribuídos modelados em Event-B.

2. REFERENCIAL TEÓRICO

Nesta Capítulo será apresentado o embasamento teórico utilizado no restante do trabalho. Ele está estruturado da seguinte forma: na Seção 2.1 será descrito o formalismo Event-B. Na Seção 2.2 será apresentado o conceito de padrões em Event-B. Na Seção 2.3 será apresentada a Plataforma RODIN. Por fim, na Seção 2.4 serão apresentados trabalhos relacionados com a proposta.

2.1 Event-B

Em Event-B um modelo é caracterizado por: um nome, uma coleção distinta de variáveis que definem os estados do sistema, invariantes, eventos e um contexto associado.

Invariantes são definições impostas a fim de restringir variáveis de um sistema assegurando uma transição para um estado correto, tornando-se necessário garantir que estas restrições serão preservadas na ocorrência de um evento. Portanto as invariantes tornam-se uma das propriedades a serem provadas para um sistema modelado em Event-B.

Por exemplo, as invariantes "inv1" e "inv2" da Figura 2.1, determinam, como propriedades de um sistema qualquer, que o número de pessoas sempre devem ser menor ou igual ao número de lugares disponíveis e ainda ser um valor natural.

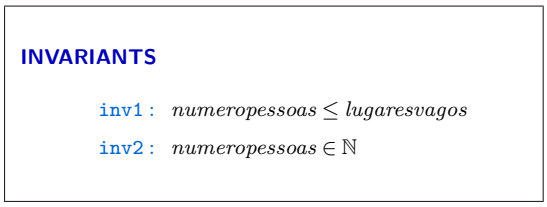


Figura 2.1 – Exemplo de invariante

Os eventos anteriormente citados, definem um conjunto de transições no sistema que podem ocorrer quando determinadas circunstâncias forem satisfeitas. Os eventos são compostos por três elementos: nome do evento ("Entrada" da Figura 2.2), guarda ("grd1" e "gr2" da Figura 2.2) e ação ("act1" da Figura 2.2).

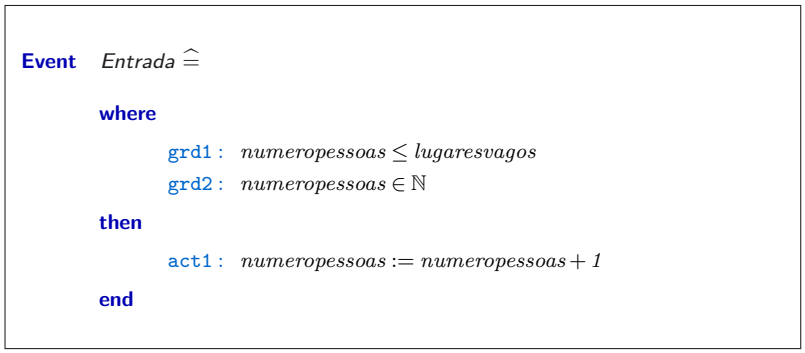


Figura 2.2 – Exemplo de evento

As guardas são predicados construídos sobre os estados das constantes e variáveis do sistema, sendo as condições necessárias para um evento ocorrer. Quando a condição de uma guarda é satisfeita, realiza-se uma ação no sistema. Ação que define o estado resultante da transição associada ao evento. Para estas transições ocorrerem são realizadas substituições nas variáveis de estado presente no modelo, sendo essas chamadas de substituições generalizadas (*generalized substitution*).

Substituições generalizadas podem estar associadas a um *before-after predicate*, onde existe uma relação entre o valor do estado antes do evento determinado acontecer, e depois do evento acontecido (n'=n+1).

Event-B não se limita apenas a modelos, existe a definição de outro componente na linguagem chamado de contexto, que define a forma como um determinado modelo vai ser parametrizado para ser instanciado.

Um contexto é composto por: um nome, uma lista distinta de *carrier sets*, uma lista distinta de constantes e de nomes das propriedades. Os *carrier sets* existentes são representados por um nome no contexto, sendo os diferentes nomes tratados independentemente. As constantes existentes no contexto são definidas por propriedades que devem ser adicionadas no contexto.

Importante observar que os contextos podem ser referenciados por modelos (Figura 2.3), quando acontece é dito que um modelo "enxerga" um contexto, sendo assim todas as *carrier set* e constantes do contexto estão disponíveis para o modelo associado a ele. A Figura 2.3 mostra as relações existentes entre modelo, contexto e eventos.

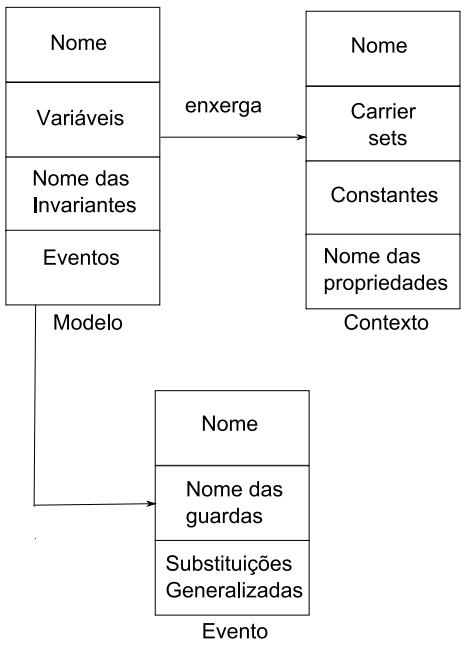


Figura 2.3 – Estrutura de modelo, evento e contexto

A aplicação de métodos formais em sistemas complexos produz um grande volume de modelos e provas, tornando-se muito trabalhosa a sua visualização e a garantia que as propriedades do sistema estão sendo asseguradas. O formalismo Event-B possui alguns conceitos especiais visando facilitar a aprendizagem e o entendimento dos sistemas que estão sendo modelados, proporcionando assim

uma melhor visualização das suas provas e propriedades.

Na Seção 2.1.1 será apresentado o conceito mais utilizado na linguagem Event-B: o refinamento.

2.1.1 Refinamento

A modelagem de um sistema, representando-se todas as propriedades e estados em apenas um nível, resultaria em um modelo único gigantesco de difícil compreensão e até mesmo uma difícil leitura [IR08].

Para solucionar esse problema, a linguagem Event-B utiliza-se do conceito de refinamento, aonde através do qual as propriedades dos sistemas se tornam mais detalhadas e são comprovadas a cada passo da modelagem, facilitando sua compreensão. O sistema é modelado incrementalmente herdando-se as propriedades dos modelos anteriores [AH07] . A cada passo de refinamento, restringe-se o comportamento do modelo, através do fortalecimento das guardas, resultando em um modelo mais preciso, determístico e mais perto da realidade.

Pode-se visualizar um exemplo de refinamento na Figura 2.4, onde um modelo N chamado de modelo concreto refina um modelo M chamado de modelo abstrato.

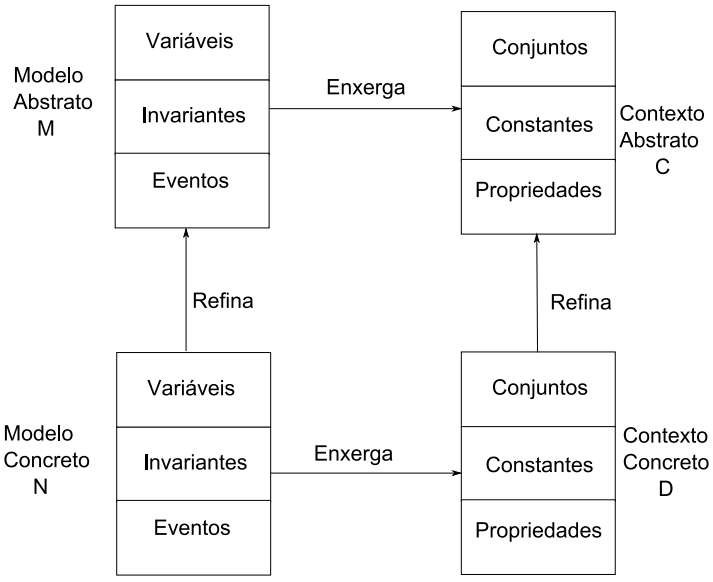


Figura 2.4 – Refinamento em Event-B [MAV05]

No exemplo da Figura 2.4 é possível perceber que um modelo concreto N "enxerga" um contexto concreto D, que refina um contexto abstrato C, que é "visto" pelo modelo abstrato M.

Os refinamentos de modelos e contextos possuem algumas diferenças. Para o refinamento do contexto da Figura 2.4, os conjuntos e constantes do contexto abstrato C são mantidos no seu refinamento e o refinamento do contexto concreto D consiste em adicionar novas constantes e propriedades (*carrier sets*) às constantes e propriedades (*carrier sets*) mantidas no contexto abstrato C.

No modelo concreto N existem as *gluing invariant* (invariantes de aglutinação), que são um conjunto de invariantes as quais consistem em: invariantes do novo modelo N (Figura 2.4), agluti-

nada às invariantes do modelo abstrato M (Figura 2.4) que ainda deverão ser conservadas. Sendo assim as *gluing invariant* relacionam as variáveis do modelo abstrato com novas variáveis do modelo concreto (refinamento).

É possível existir um número muito grande de refinamentos, porém é importante salientar-se que as *gluing invariant* ocorrem em apenas dois níveis de refinamento, portanto apenas aglutinando o modelo concreto e abstrato que estará sendo tratado em questão.

Refinamentos em Event-B tratam diferentemente eventos existentes e eventos adicionados naquele passo de refinamento.

Eventos existentes, quando refinados, devem se adequar as variáveis adicionadas naquele passo de refinamento, sendo necessário garantir que seu refinamento realizou-se de modo correto. Em eventos do modelo concreto, que possuem uma mesma substituição generalizada (mesma ação) é possível realizar a união dos eventos, unindo suas guardas e assim transformando-se "n" eventos em apenas um evento no modelo concreto, como demonstra a Figura 2.5.

Eventos Separados	União de Eventos
E = when G(v) then S(v) end F = when H(v) then S(v) end	EF = when G(v) V H(v) then S(v) end

Figura 2.5 – União de Eventos

Entretanto para refinamento de novos eventos há duas restrições: o novo evento deve refinar um evento *skip* (evento vazio) do modelo anterior, e deve-se permitir que cada guarda em algum momento possa ser verdadeira.

Todas as restrições, assim como as propriedades que o sistema deve preservar valida-se através das *proof obligations*.

2.1.2 *Proof Obligations*

Proof obligations (obrigações de prova) é o que caracteriza a semântica em Event-B, assim como verifica as propriedades contidas no modelo, transformando a linguagem Event-B em um cálculo para modelagem que independe dos vários modelos de computação. Confia-se nesta uniformidade como chave para o ensino de diferentes aspectos da modelagem de sistemas [Hal06].

Quando um sistema é modelado em Event-B, o *before-after predicate* é responsável pela transição de estado do sistema (v para v' da Figura 2.6). Para modelagem correta do sistema é necessário uma *proof obligation* que garanta possibilidade de transição. Esta propriedade é garantida através da *proof obligation* FIS (*feasibility statement*) .Tendo v como a representação do conjunto das variáveis do modelo, a FIS garante que as propriedades do modelo (P), as invariantes (I) e as guardas (G) vão implicar na existência de um estado futuro (R) (Figura 2.6).

Os possíveis estados de um sistema modelado em Event-B são determinados pelas invariantes, ou seja, se as invariantes do sistema são mantidas, então o estado é válido. Para garantir essa

Propriedades Invariantes Guardas ⊢ $\exists v' \cdot$ *before-after predicate*	P(v) I(v) G(v) ⊢ $\exists v' \cdot$R(v, v')

Figura 2.6 – *proof obligation* FIS

propriedade, usa-se a *proof obligation* INV (*Invariant Preservation*) que pode ser visualizada na Figura 2.7. As propriedades do modelo (P), as invariantes (I), as guardas (G) e o *before-after predicate* (R) devem implicar em um novo estado onde as invariantes continuem sendo conservadas pelo sistema. Esta *proof obligation* deve ser realizada para cada evento.

Propriedades Invariantes Guardas *before-after predicate* ⊢ invariantes modificadas	P (v) I (v) G (v) R (v, v') ⊢ I (v')

Figura 2.7 – *proof obligation* INV

Em um modelo em Event-B é importante garantir também que o sistema não entre em *deadlock*, ou seja, que sempre exista uma guarda que possa ocorrer. A *proof obligation* que assegura esta propriedade é a DLKF (*deadlock free*) . As propriedades (P) e as invariantes (I) devem implicar em uma condição a qual uma das guardas existentes no modelo deve ser verdadeira, pode-se visualizar essa relação na Figura 2.8.

Propriedades Invariantes ⊢ $\exists \cdot$ Guarda verdadeira	P (v) I (v) ⊢ G1(v) $\vee ... \vee$ Gn(v)

Figura 2.8 – *proof obligation* DLKF

Ao ser realizado um refinamento deve-se garantir que o modelo refinado (modelo concreto) ainda conserva as propriedades do modelo anterior (modelo abstrato). Para tanto utiliza-se as *proof obligations* descritas anteriormente, com a diferença que as mesmas devem comprovar que o refinamento conserva as propriedades do modelo anteriormente provadas. Tendo v como as variáveis do modelo abstrato e w como as variáveis do modelo concreto, as *proof obligations* do refinamento que realizam essas provas são chamadas de FIS_REF , GRD_REF, INV_REF e podem ser visualizada na Figura 2.9.

Como pode ser visualizado na Figura 2.9, a proof obligation FIS_REF define que as propriedades (P), as invariantes (I), a *gluing invariant* (J) e as guardas (H - modelo concreto) devem implicar em existir um transição de estado(S). Na GRD_REF as propriedades (P), as invariantes (I), a *gluing invariant* (J), e as guardas (H - modelo concreto) devem implicar nas guardas (G) do modelo abstrato.

INV_REF descreve que: as propriedades (P), as invariantes (I), a *gluing invariant* (J), as guardas (H - modelo concreto) e o *before-after predicate* do modelo abstrato (S) devem implicar em existir

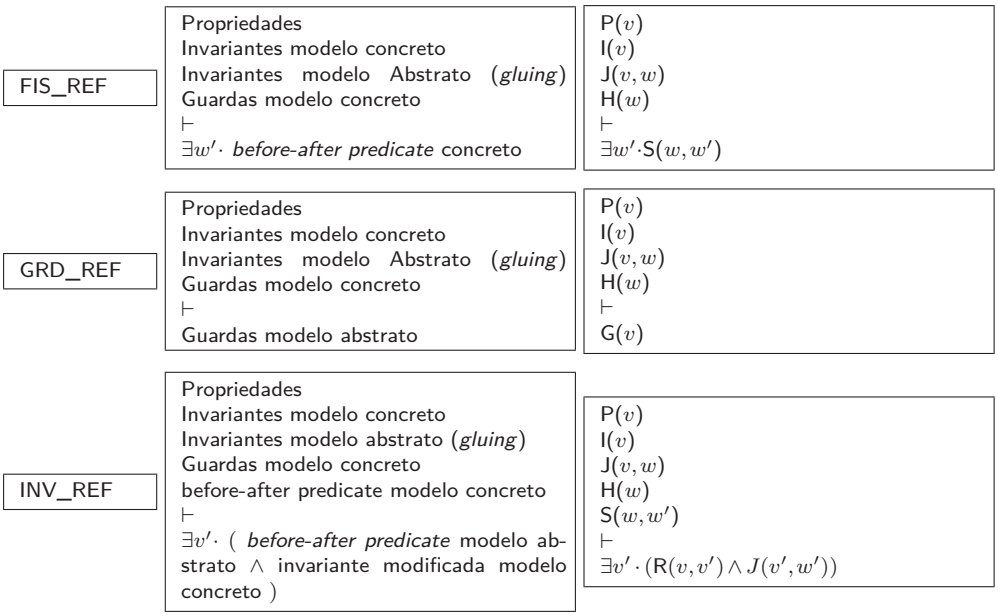


Figura 2.9 – *proof obligation* do refinamento

um estado que satisfaça o *before-after predicate* do modelo concreto (R) e a *gluing invariant* (J). Entretanto a INV_REF possui uma terceira utilização que é para novos eventos, adicionados naquele passo de refinamento. Isso deve-se ao fato que é necessário provar que os novos eventos irão conservar as invariantes do modelo abstrato.

Para tanto utiliza-se a *proof obligation* da Figura 2.10, onde as propriedades (P), as invariantes (I), a *gluing invariant* (J), as guardas do modelo concreto (H) e o *before-after predicate* (S) implicam em o novo estado que deve conservar a *gluing invariant* (J).

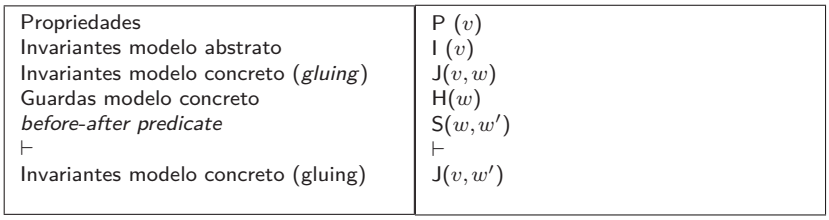


Figura 2.10 – INV para novos eventos

Quando um novo evento é adicionado na modelagem existe a possibilidade deste evento tomar o controle do sistema, assim impossibilitando que outros eventos existentes ocorram. Para que isso não ocorra, existe a *proof obligation* WFD_REF da Figura 2.11, que garante que eventos não tomem posse eternamente do sistema. Para garantir isso, deve-se encontrar uma equação chamada de variante, que os eventos decrementem com a transição de estados do sistema.

A WFD_REF é definida da seguinte maneira: as propriedades (P), as invariantes (I), a *gluing invariant* (J), as guardas (H), e o *before-after predicate* (S), devem implicar em uma variante (V) que pertença aos naturais e que em um estado futuro seja menor que no atual estado.

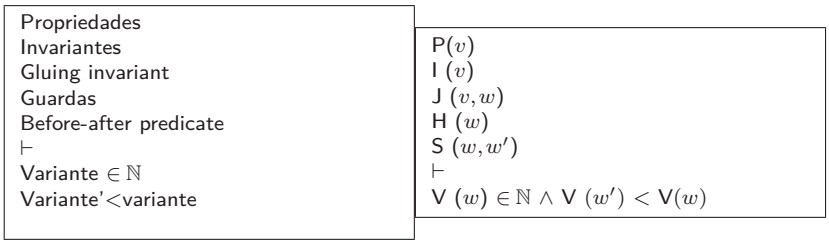


Figura 2.11 – *proof obligation* WFD_REF

2.1.3 Event-B e B

Event-B e B são métodos formais semelhantes. A abordagem de ambos métodos para demostrar a corretude do sistema é através da descarga de um conjunto de obrigações de provas (*proof obligations*). Entretanto a estrutura, assim como alguns itens presentes na linguagem do formalismo Event-B foram renomeados, diferenciando-se do método B.

No método B todas as características do modelo são descritas em uma máquina (Figura 2.12) [Sch01]. Além disso as guardas são especificadas como "pre"(pré-condição) e os eventos como "*operations*".

```
MACHINE
CONSTRAINTS
SETS
CONSTANTS
PROPERTIES
VARIABLES
INVARIANT
INITIALISATION
OPERATIONS
      name_operation
         PRE
            |
         THEN
         END ;
END
```

Figura 2.12 – Máquina em B.

Em Event-B existe a separação da parte estática (contexto) e a parte dinâmica (modelo) do sistema modelado (Figura 2.3), ficando em diferentes estruturas as constantes e suas propriedades, assim como as variáveis do sistemas e as invariantes. Além disso, Event-B possui uma ferramenta que ajuda no desenvolvimento da modelagem e que concentra-se na geração das *proof obligation* e dedução automática das *proof obligation* triviais [IR08]. Desta forma, tentando deixar apenas para o desenvolvedor a preocupação com a modelagem e não com suas provas. Esta ferramenta é implementada no projeto RODIN e é uma extensão da plataforma Eclipse (Seção 2.3).

Pelas dificuldades e complicações que poderia existir no aprendizado do formalismo Event-B na ferramenta, os desenvolvedores adotaram a terminologia "máquina" [Hal06] ao invés de "modelos" [MAV05] e "axiomas" ao invés de "propriedades", diferenciando-se portanto do manual da linguagem. Segundo Hallerstede em [Hal06], ficaria confuso utilizar-se da terminologia modelos em uma frase

como: "um modelo consiste de modelos e contextos".

Desta forma ficando contextos e máquinas[1] caracterizados da seguinte forma: os contextos da plataforma Event-B possuem *carrier sets*, constantes, axiomas e teoremas; Já as máquinas possuem variáveis, invariantes, teoremas, eventos e variantes.

2.2 Padrões em Event-B

Padrões são construções que permitem o reuso de soluções genéricas para problemas frequentes, sendo extensivamente discutidos na engenharia de software há vários anos [GHJV95].

O emprego da noção de padrões a especificações formais permite o reuso de artefatos custosos, dado que formalmente especificados e analisados, integrando-os a sistemas alvo desejados. Além de promover o reuso em fases mais iniciais do desenvolvimento, de grande importância é o reuso também de provas já realizadas sobre o padrão. Estas não precisam ser repetidas sobre o sistema alvo.

Em métodos que suportam a noção de refinamento, o desenvolvimento se dá por uma cadeia de modelos relacionado. A cada passo de refinamento, novas propriedades são adicionadas ao modelo. Neste contexto, um padrão guia um passo de refinamento de tal forma que o modelo concreto obtido apresente um conjunto de propriedades definidas e provadas no padrão. Para Event-B, este processo é discutido tanto em [Für09] como em [Ili07].

Assim, um padrão em Event-B é composto de um modelo abstrato P_A, chamado em [Für09] de especificação padrão, e de um modelo concreto P_C, chamado de refinamento padrão, definindo um passo consistente de refinamento: $P_A \sqsubseteq P_C$. Juntamente com este passo de refinamento, propriedades do padrão são provadas.

Para o emprego de um determinado padrão em um sistema alvo são necessários de três fases [Für09]: a combinação, a verificação (Seção 2.2.2) e a incorporação (Seção 2.2.3).

2.2.1 Combinação

A primeira fase é responsável por relacionar a especificação padrão (P_A) e o modelo do sistema alvo (chamado nesta descrição de $refinamento\ n$). Nesta fase são especificadas as variáveis e os eventos do modelo alvo que serão relacionadas com o padrão. Para tanto, esta fase, é constituida de dois passos: a definição das variáveis combinadas (C1) e a definição dos eventos combinados (C2).

Tendo A e B como variáveis da especificação padrão (P_A) e C,D e E variáveis do modelo do sistema alvo ($refinamento\ n$), no passo C1, combina-se uma a uma as variáveis da especificação padrão com as variáveis do modelo alvo ($refinamento\ n$) (linha 1 e 2 da Tabela 2.1). Usualmente, a especificação padrão, pode conter um menor número de variáveis que o modelo alvo, existindo variáveis no modelo alvo não combinadas. Essas variáveis são chamadas *variáveis extras* (linha 3 da Tabela 2.1).

[1] neste trabalho são usado ambos os nomes: máquina e modelo

Tabela 2.1 – C1

	Variáveis Combinadas:
1	$A \rightsquigarrow C$
2	$B \rightsquigarrow D$
	Variáveis não Combinadas (*variáveis extras*):
3	E

Tendo $event_{PA}$ como evento da especificação padrão (P_A) e os eventos $event1_{REFn}$ e $event2_{REFn}$ do modelo alvo ($refinamento\ n$), no passo C2, indica-se os eventos que serão combinados (linha 1 da Tabela 2.2). Na combinação dos eventos, igualmente ao acontecido com a combinação das variáveis, existe a probabilidade de o modelo alvo possuir *eventos extras*, ou seja, eventos não combinados (linha 2 Tabela 2.2).

Tabela 2.2 – C2

	Eventos Combinados:
1	$event_{PA} \rightsquigarrow event1_{REFn}$
	Eventos não Combinados (eventos extras):
2	$event2_{REFn}$

Concluída a primeira fase, todas as variáveis e eventos da especificação padrão (P_A) devem conter uma combinação no modelo alvo ($refinamento\ n$). Entretanto a combinação dos eventos e das variáveis não garantem que o padrão é realmente compatível com o modelo alvo. Portanto sendo necessário verificar se a combinação é correta.

2.2.2 Verificação

Na segunda fase é preciso validar a combinação realizada na primeira fase, ou seja, verificar se a especificação padrão (P_A) é compatível com o modelo alvo ($refinamento\ n$). Para tanto, os eventos combinados do modelo alvo ($refinamento\ n$), deverão ser verificados um a um a sua compatibilidade com o padrão. A verificação dos eventos combinados acontece da seguinte forma:

V1 As guardas da especificação padrão devem ser sintaticamente iguais as guardas do modelo alvo.

V2 Deve-se verificar as existências de *guardas extras*, ou seja, guardas não combinadas do modelo alvo.

V3 As ações das especificação padrão devem ser sintaticamente iguais as ações do modelo alvo.

V4 Deve-se verificar as existências de *ações extras*, ou seja, ações não combinadas do modelo alvo.

Para os eventos não combinados do modelo alvo, *eventos extras*, apenas deve-se verificar se eles não alteram alguma variável combinada (V5). Sendo respeitadas estas condições o padrão pode ser aplicando no modelo alvo, passando-se para última fase: a incorporação.

2.2.3 Incorporação

A fase de incorporação é caracterizada como um passo de refinamento do modelo alvo. Através da incorporação as características do padrão são introduzidas no modelo. Para tanto, diferentemente das fases de combinação e verificação, utiliza-se o refinamento padrão (P_C). Através da incorporação, o modelo alvo fica constituído de um modelo abstrato ($refinamento\ n$) e um modelo concreto (chamado nesta descrição de $refinamento\ n+1$).

Para a modelagem do $refinamento\ n+1$ (incorporação do padrão), primeiro deve-se copiar para o modelo as variáveis do refinamento padrão e as variáveis não combinadas do modelo abstrato ($refinamento\ n$) (passo chamado I1).

Logo após copiado as variáveis para o modelo concreto ($refinamento\ n+1$), no passo chamado I2, realiza-se a construção das invariantes do modelo $refinamento\ n+1$. Este passo é responsável por adicionar as propriedades preservadas do padrão. Para tanto, as invariantes do $refinamento\ n+1$ são construídas apartir das invariantes da especificação do padrão (P_A) e do refinamento do padrão (P_C).

Neste ponto da incorporação, o modelo concreto ($refinamento\ n+1$) contém as invariantes e as variáveis necessárias ao modelo, restando apenas adicionar os eventos.

A modelagem dos eventos no $refinamento\ n+1$ é divida em três tipos de eventos: eventos não combinados do $refinamento\ n$ (passo I3), eventos criados no refinamento padrão (P_C) que não refinam eventos da especificação padrão (P_A)(passo I4) e por último os eventos combinados (passo I5).

Os *eventos extras*, ou seja, eventos não combinados do modelo abstrato ($refinamento\ n$) e os eventos criados no refinamento padrão (P_C) devem ser copiados um a um para o $refinamento\ n+1$. Entretanto os pares de eventos combinados, fase C2 da Seção 2.2.1, possuem um tratamento diferenciado na incorporação:

- Inicialmente, copia-se para o $refinamento\ n+1$ as guardas do evento combinado presente no refinamento padrão (passo I5.1).
- Depois adiciona-se, nas guardas anteriormente copiadas, as *guardas extras* provenientes do evento combinado do modelo abstrato (passo V2 da Seção 2.2.2) (passo I5.2).
- Concluído a construção das guardas do evento, copia-se para o mesmo, as ações do evento combinado presente no refinamento padrão (P_C) (passo I5.3).
- A seguir adiciona-se, nas ações anteriomente copiadas, as *ações extras* provenientes do evento combinado do modelo abstrato (V4 da Seção 2.2.2) (passo I5.4).

Como visto anteriormente, no passo I1, são copiadas as variáveis combinadas do refinamento padrão para o modelo concreto ($refinamento\ n+1$). Esta ação faz com que as variáveis combinadas do modelo abstrato ($refinamento\ n$) desapareçam no modelo concreto ($refinamento\ n+1$). Por razões de *meta-proof* [Für09], no modelo concreto ($refinamento\ n+1$), as variáveis

do refinamento padrão, identificadas na fase C1 da Seção 2.2.1, devem ser renomeadas por sua respectiva combinação do modelo abstrato ($refinamento\ n$).

Outra opção é a inclusão de uma invariante no $refinamento\ n+1$ (modelo concreto). Esta invariante teria como finalidade identificar as variáveis que desapareceram do modelo abstrato ($refinamento\ n$), nas variáveis oriundas do refinamento padrão (P_C).

Torna-se importante salientar que as modificações, anteriomente citadas, devem ser refletidas no restante do modelo, ou seja, é necessário refletir as modificações das variáveis nos eventos e invariantes.

No Capítulo 4 será apresentado em detalhe a aplicação de um padrão em um sistema distribuído.

2.3 Plataforma RODIN

A utilização de um modelo formal, para o desenvolvimento de sistemas complexos, como visto anteriormente, torna-se uma tarefa necessária a fim de garantir que um sistema vai ser bem arquitetado e provado. Para tanto necessita-se uma boa ferramenta como suporte para especificação da notação de modelagem.

Para o formalismo Event-B, existe a plataforma RODIN[2] como suporte para construção e validação das propriedades desejadas nos modelos. A plataforma RODIN foi implementada no projeto RODIN[3] (*Rigorous Open Development Environment for Complex Systems*), por se tratar de uma ferramenta aberta. Desta forma facilitando assim a sua adaptação e extensão para *plug-ins*[4] como novos provadores de teoremas e ferramentas de análise de model checker [BH07].

Na plataforma RODIN, as obrigações de provas (*proof obligations*) são descarregadas automáticamente, proporcionando um rápido *feedback* para o desenvolvedor sobre construção e a preservação das propriedades do modelo. Além disso, é possível selecionar o conjuntos de provas que serão descarregados automáticamente nos modelos Event-B.

Dadas estas características, nota-se que a plataforma RODIN fornece um apoio considerável na modelagem de sistemas em Event-B.

2.4 Trabalhos Relacionados

Nesta seção são apresentados trabalhos relacionados com a modelagem de sistemas distribuídos em Event-B.

Em [YB06] é apresentada a modelagem e análise em Event-B de um mecanimo de transação distribuída em uma base de dados. O problema distribuído modelado, baseia-se em garantir que as diferentes base de dados estejam consistentes. Para garantir essa integridade são utilizados protocolos *commit* [GR92], através do qual é possivel garantir que todas as bases de dados realizem o *commit* (aceitação) de uma transação ou abortem caso existam multiplas falhas.

[2]RODIN Platform http://www.event-b.org/ (último acesso em dezembro 2009).
[3]Project RODIN http://rodin.cs.ncl.ac.uk/ (último acesso em dezembro de 2009)
[4]Plug-ins RODIN Platform http://www.event-b.org/plugins.html (último acesso em dezembro 2009).

Em [Abr01] é possivel encontrar modelado em Event-B uma grande variedade de sistemas distribuídos, dentre os quais se encontram: o algoritmo de eleição de líder em topologia anel, o algoritmo de roteamento distribuído para agentes movéis e o algoritmo de exclusão mútua distribuída.

Em [CM06] é modelado um algoritmo distribuído de contagem de referência (DRC) em Event-B. O algoritmo de DRC é utilizado para compartilhar recursos de forma distribuída e remover os mesmos caso não estejam mais sendo utilizados. A remoção é realizada de modo *distributed garbage collection*[5].

Em [MAV05] existe a modelagem de um protocolo clássico existente, o *two-phase handshake*. O *two-phase handshake* é um protocolo composto por um *sender* responsável por enviar mensagens e um *receiver* responsável por receber a as mesmas. O Protocolo é baseado no principio da transferência de pacotes de dados entre dois dispositivos, sendo que para isso é necessária uma ordem para as entregas de pacotes do *sender* ao *receiver*.

A seguir são apresentados em detalhes, a modelagem de dois sistemas distribuídos em event-B: o algoritmo distribuído de contagem de referência e o protocolo *two-phase handshake*.

Algoritmo Distribuído de Contagem de Referência

No algoritmo de DRC um conjunto finito de *sites* interagem de maneira distribuída, cada qual pode criar o seu recurso e compartilhá-lo com outros *sites*, sendo que o criador do recurso é chamado de proprietário. Os recursos não precisam apenas ser compartilhados a partir do *owner* (proprietário), os *sites* que possuem cópias do recurso do proprietário também podem compartilhar esses recursos com outros *sites*.

Entretanto um recurso só poderá ser removido do proprietário quando o mesmo não estiver sendo utilizado por qualquer outro *site*. Existindo a possibilidade de *sites* poderem compartilhar um recurso sem ser o *site* proprietário, acaba-se tornando importante o proprietário do mesmo ter conhecimento de quantos *sites* estão a utilizar o seu recurso.

O algoritmo proposto por Moreau and Duprat [MD01] ocorre da seguinte maneira: O *site* proprietário manda seu recurso para o *site* $s1$ (I da Figura 2.13); o *site* $s1$ propaga o recurso do site proprietário para o *site* $s2$ (II da Figura 2.13); o *site* $s1$ incrementa sua variável, representando a propagação do recurso, sem o conhecimento do site proprietário (II da Figura 2.13); o *site* $s2$ recebe o recurso do *site* $s1$ e manda uma mensagem para o *site* proprietário avisando que recebeu o seu recurso (III da Figura 2.13); o *site* proprietário incrementa sua variável e manda uma mensagem para o *site* $s1$, avisando que foi informado que um novo *site* possui seu recurso (IV da Figura 2.13); o *site* $s1$ decrementa sua variável (V da Figura 2.13).

A mensagem enviada do *site* $s1$ para o *site* $s2$ chama-se *send*. A mensagem do *site* $s2$ para o *site* proprietário é a *inc_dec*. E a última mensagem enviada, do proprietário para o $s1$ é chamada de *dec*. O funcionamento do algoritmo pode ser visualizado na Figura 2.13.

O Envio de uma mensagem assim como a recepção da mesma, nesse sistema modelado em

[5]distribuição onde referencias a objetos são realizadas por clientes remotos

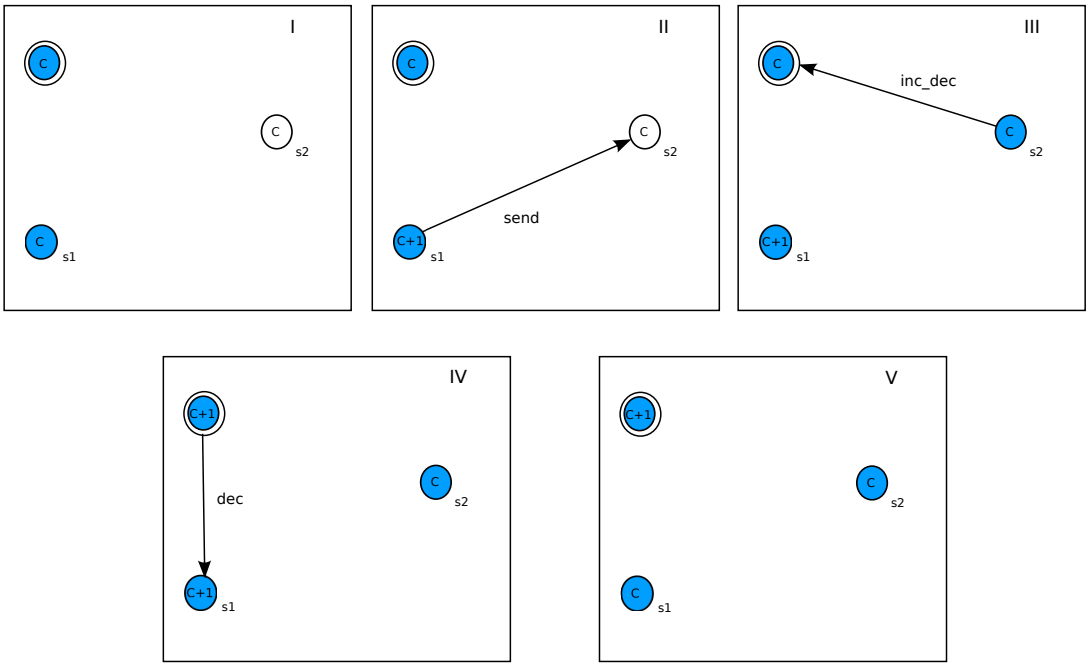


Figura 2.13 – algoritmo de DRC [CM06]

Event-B, foi representado por eventos. Quando a mensagem é enviada de um dado *site* para outro, o evento responsável inclui ambos os *sites* em uma variável, representando assim o envio da mensagem.

Para recepção da mensagem no segundo *site* é utilizado outro evento, o qual remove da variável que estava presente no evento de envio os dados que simbolizavam o processo de envio da mensagem. Em [CM06] existem eventos responsáveis pelas trocas de mensagens, eventos como: $send\_copy$, $receive\_copy$, $receive\_inc$ e $receive\_sendDec$ que serão descritos a seguir.

Destes eventos,citados anteriomente e modelados em Event-B, o primeiro (Figura 2.14) é responsável pelo envio de uma mensagem do tipo *send*, o segundo (Figura 2.15) pela recepção de uma mensagem do mesmo tipo e o envio de uma mensagem tipo *inc*.

Para o evento da Figura 2.14 ocorrer as seguintes guardas precisam ser verdadeiras: O *site* $s1$ precisa ter o recurso (linha 3 da Figura 2.14); o *site* $s2$ precisa estar no conjunto dos *sites* que podem possuir o recurso disponível, conjunto este representado pela variável $SITES$ (linha 4 da Figura 2.14); por último o contador c deve pertencer ao conjunto $COUNT$, mas não deve ter sido indexado a nenhuma outra mensagem (linha 5 da Figura 2.14).

1 $send\_copy \mathrel{\widehat{=}}$
2 **any** $s1$,$s2$,c **where**
3 $s1 \in REC$
4 $s2 \in SITES$
5 $c \in COUNT - count$
6 **then**
7 $Send := Send \cup \{c \mapsto (s1 \mapsto s2)\}$
8 $SendLoc := SendLoc \cup \{c \mapsto (s1 \mapsto s2)\}$
9 $count := count \cup \{c\}$
10 **end**

Figura 2.14 – Evento de envio de mensagem *send* [CM06]

Os *sites* que possuem o recurso do proprietário são representados pelo conjunto da variável REC.

Se a guarda for satisfeita então o evento *send_copy* realiza as seguintes ações: adiciona ao conjunto da variável $Send$ a mensagem enviada de $s1$ para $s2$, indexada ao valor do contador c (linha 7 da Figura 2.14); adiciona os mesmos valores para o conjunto da variável $SendLoc$ (linha 8 da Figura 2.14); adiciona o contador ao conjunto da variável $count$, o qual representa as diferentes mensagens indexadas (linha 9 da Figura 2.14).

A variável $SendLoc$ é o conjunto de mensagens de propagação de recurso que não tiveram conhecimento do proprietário. Para remover uma dado desta variável é necessário o proprietário enviar uma mensagem de $SendDec$ para o *site* $s1$, assim o proprietário indica ao *site* $s1$ que obteve conhecimento de seu recurso ter sido propagado por ele.

Para o evento $receive\_copy$ ocorrer e o *site* $s2$ receber a mensagem, as seguintes guardas deveram ser satisfeitas: deve existir uma mensagem enviada de $s1$ para $s2$ no conjunto da variável $Send$ (linha 3 da Figura 2.15); O $s2$ não deve possuir ainda o recurso (linha 4 da Figura 2.15).

```
1 receive_copy ≙
2 any s1, s2, c where
3 c ↦ (s1 ↦ s2) ∈ Send
4 s2 ∉ REC
5 then
6 Inc := Inc ∪ {c ↦ (s1 ↦ s2)}
7 Send := Send − {c ↦ (s1 ↦ s2)}
8 REC := REC ∪ {s2}
9 end
```

Figura 2.15 – Evento de recebimento de uma mensagem *send* [CM06]

Caso as guardas sejam verdadeiras o evento adiciona uma mensagem no conjunto da variável Inc (linha 6 da Figura 2.15), e remove a respectiva mensagem do conjunto da variável $Send$ (linha 7 da Figura 2.15), indicando assim que a mensagem foi entregue. O $s2$ é adicionado ao conjunto da variável REC, portanto recebendo o recurso do proprietário (linha 8 da Figura 2.15). O evento da Figura 2.16 é responsável por receber a mensagem de Inc.

```
1 receive_Inc ≙
2 any s1, s2, c where
3 c ↦ (s1 ↦ s2) ∈ Inc − Dec
4 then
5 Inc := Inc − {c ↦ (s1 ↦ s2)}
6 RecOwn := RecOwn ∪ {c ↦ (s1 ↦ s2)}
7 SendDec := SendDec ∪ {c ↦ (s1 ↦ s2)}
8 end
```

Figura 2.16 – Evento *receive_inc* [CM06]

A variável Dec da Figura 2.16 é responsável por gerenciar as *dec* mensagens. A operação $Inc - Dec$ (linha 3 da Figura 2.16) resultará exatamente da mensagem inc_dec do algoritmo proposto por Moreau and Duprat, sendo que a relação dos *sites* indexada ao contador c devem pertencer a este conjunto para o evento $receive\_Inc$ ocorrer (linha 3 da Figura 2.16). Sendo a guarda verdadeira, a relação deste *sites* é removida da variável Inc (linha 5 da Figura 2.16),

adicionada na variável $RecOwn$ (linha 6 da Figura 2.16) e $SendDec$ (linha 7 da Figura 2.16). A variável $RecOwn$ é o conhecimento local do proprietário sobre o *status* do seu recurso. Então chegado ao evento *receive_inc* (Figura 2.16) o proprietário possui conhecimento da sua cópia ter sido propagada, agora restando ao propagador ($s1$) receber a mensagem de *SendDec* enviada pelo proprietário. Fato que é ocorrido no evento da Figura 2.17.

1 $receive\_SendDec \mathrel{\widehat{=}}$
2 **any** $s1$, $s2$, c **where**
3 $c \mapsto (s1 \mapsto s2) \in SendDec$
4 **then**
5 $SendDec := SendDec - \{c \mapsto (s1 \mapsto s2)\}$
6 $SendLoc := SendLoc - \{c \mapsto (s1 \mapsto s2)\}$
7 **end**

Figura 2.17 – Evento *receive_sendDec* [CM06]

Portanto para o evento da Figura 2.17 ocorrer, a relação dos *sites* indexados ao contador deve existir no conjunto da variável $SendDec$ (linha 3 da Figura 2.17). Tornando-se verdadeira esta guarda, esta relação é retirada do conjunto da variável $SendDec$ (linha 5 da Figura 2.17), simbolizando a entrega da mensagem ao *site* $s1$. O mesmo ocorre no conjunto da variável $SendLoc$ (linha 6 da Figura 2.17), simbolizando o decremento do propagador no algoritmo de Moreau and Duprat.

As propriedades distribuídas foram incrementalmente inseridas, sendo que de início foi preciso apenas garantir as propriedades de cada elemento em separado e depois ir adicionando aos poucos complexidade do problema. O mesmo ocorre em [MAV05], onde existe a modelagem de um protocolo clássico existente, o *two-phase handshake*.

Protocolo Two-Phase Handshake

O protocolo *two-phase handshake* é baseado no envio de pacotes de dados entre dois processos em apenas um sentido. Desta forma, o processo *receiver*, apenas tem a responsabilidade da confirmação das mensagens enviadas pelo processo *sender*, assim tornando-se possível manter controle sobre as mensagens recebidas, e havendo um reenvio caso necessário. Para tanto é necessário dois canais de comunicação entre os dispositivos, um responsável pelo envio dos dados e outro pela confirmação do recebimento.

Para modelagem deste protocolo em Event-B foi utilizado de um princípio semelhante do encontrado em [CM06], onde uma variável do modelo representa o canal de comunicação e eventos são responsáveis pelo envio e recebimento das mensagens.

Em [MAV05], o modelo em Event-B no início era apenas constituído de um evento responsável por uma atribuição de variáveis $(g := f)$ e suas propriedades e invariantes a serem provadas em cima dessa atribuição. A Cada passo de refinamento o modelo foi incrementando com características do sistema real, se tornando cada vez mais concreto e percebendo-se a concretização do *sender* e *receiver*.

Entretanto nesta modelagem diferentemente da existente em [CM06] chegou-se em ponto final onde foi utilizado de decomposição do modelo formando-se dois modelos, um contendo o *sender* e outro o *receiver*. Os Eventos existentes nos mesmos responsáveis pela transferência dos dados podem ser visualizados na Figura 2.18.

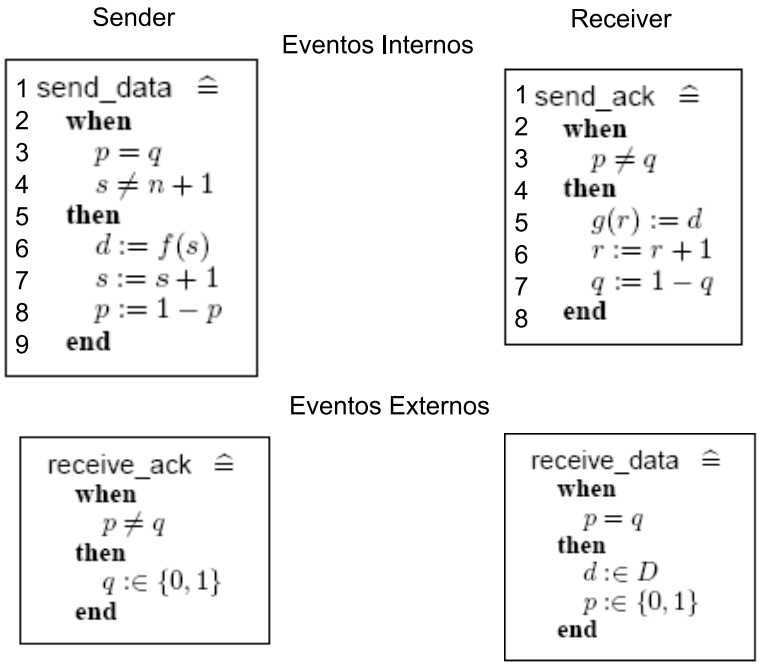


Figura 2.18 – Eventos do *sender* e *receiver*

As variáveis p e q na Figura 2.18, são variáveis externas do modelo, ou seja, após a decomposição do modelo tornaram-se necessárias ao $sender$ e o $receiver$. As variáveis externa p e q, composta por um *bit* de paridade (0 e 1), representam: o valor da última mensagem enviada (p); o valor da última mensagem recebida (q).

A variável s é o número do pacote que está sendo enviado, sendo assegurada pela variável n que representa a quantidade de pacotes.

Para o evento $send\_data$ ocorrer, ou seja, o *sender* enviar um pacotes de dados, às seguintes guardas devem ser verdadeiras: a última mensagem enviada pelo *sender* deve ter sido recebida (linha 3 da Figura 2.18) e a quantidade de pacotes enviados não pode exceder a quantidade de pacotes disponíveis (linha 4 da Figura 2.18).

Sendo verdadeira a guarda, o *sender* coloca o pacote a ser enviando na variável externa d que representa o canal de comunicação entre o $sender$ e o $receiver$ (linha 6), incrementa-se o s (linha 7) simbolizando que mais um pacote foi enviado e o bit de paridade de p é trocado (linha 8). Portanto enquanto q não voltar a ser igual a p o canal de comunicação continuará a conter o mesmo pacote.

Para o q voltar a ser igual a p, o *receiver* deve receber o dado contido no canal, necessitando que o evento $send\_ack$ (Figura 2.18) ocorra. Para isso a seguinte guarda deve ser satisfeita: o bit de paridade do ultimo dado enviado deve ser diferente do recebido (linha 3 da Figura 2.18). Sendo verdadeira a guarda, o *receiver* recebe o pacote contido no canal (linha 5 da Figura 2.18), o número

do pacote recebido é incrementado (linha 6 da Figura 2.18) e o bit de paridade de q é modificado simbolizando que a mensagem foi entregue (linha 7 da Figura 2.18).

3. BIBLIOTECA DE PADRÕES PARA SISTEMAS DISTRIBUÍDOS

Com a modelagem dos sistemas distribuídos em Event-B, presentes no apêndice A deste trabalho, percebeu-se a necessidade de repetir semânticas de comunicação nos diversos modelos. A possibilidade do reuso das semânticas de comunicação tornou-se atrativa.

Neste capítulo apresenta-se uma biblioteca de especificações em Event-B. Esta biblioteca tem como objetivo apresentar modelos que permitam o reuso de diversas semânticas de comunicação. Desta forma, libera-se o desenvolvedor que estiver modelando um sistema distribuído em Event-B das preocupações quanto a validação da comunicação do seu sistema.

Durante o desenvolvimento de sistemas distribuídos, baseados em troca de mensagens, deve-se ter uma definição clara das suposições sobre o mecanismo subjascente de troca de mensagens utilizado.

Tais suposições podem implicar em diferenças significativas no sistema em desenvolvimento. A literatura reporta várias características clássicas, de mecanismos de troca de mensagens, que podem ser assumidas para a construção de um sistema. As seguintes são consideradas neste trabalho:

- envio de mensagens: *unicast* e *multicast*;
- ordenação diferenciada para comunicações *multicast* e *unicast*:
  - *unicast*: tem-se apenas as possibilidades de entrega ordenada ou não;
  - *multicast* (comunicação em grupo): as ordens contempladas pela biblioteca são: total, FIFO (*First In First Out*) e causal;
- primitivas de comunicação: síncronas ou assíncronas;
- em algumas situações, a entrega pode acontecer com duplicatas.

A comunicação *unicast* caracteriza-se pelo envio de uma dada mensagem M de um processo E para um processo R, ou seja, trata-se de uma comunicação um-para-um. Por outro lado, a comunicação *multicast* caracteriza-se pelo envio de uma dada mensagem M de um processo E para um grupo de processos GP, ou seja, trata-se de uma comunicação realizada de um-para-muitos.

A comunicação *multicast* exige coordenação e acordo entre os processos (emissor e receptores). O objetivo deste tipo de comunicação é que cada processo receptor, receba cópias das mensagens enviadas para o grupo, com garantias de distribuição. Estas garantias incluem o acordo sobre o conjunto de mensagens que cada processo deve receber e a ordem de entrega para os membros do grupo [DKC05].

Na ordenação *multicast* FIFO, as mensagens de um mesmo processo emissor E devem chegar para os processos do grupo *multicast* GP na mesma ordem de envio. A ordem de recepção de mensagens é coerente com a ordem de envio de mensagem de um mesmo originador.

A ordenação *multicast* causal entrega as mensagens, para os processos do grupo *multicast* GP, de modo que a pontecial causalidade entre mensagens diferentes seja preservadas. Ou seja, se uma mensagem $m1$ preceder uma mensagem $m2$ por causalidade, para todo processo receptor, a mensagem $m2$ deverá ser entregue antes da $m1$.

No *multicast* totalmente ordenado atômico, neste trabalho chamado de *multicast* de ordem total, todas as mensagens são recebidas na ordem do seu envio, independente de originador e destinatários. Para tal, supõe-se a existência de relógio global. Dado este fator, torna-se importante salientar que esta ordenação é considerada fictícia. Na prática torna-se impossível manter um relógio global entre os diferentes processos emissores.

Na comunicação síncrona, o processo emissor fica bloqueado até saber que sua requisição foi aceita pelo receptor. Já na comunicação assíncrona, o emissor continua sua execução imediatamente após ter enviado sua mensagem, não ficando bloqueado [TS07].

Configurações significativas das características anteriormente citadas foram consideradas para representação da biblioteca de especificações formais de padrões para mecanismos de troca de mensagens. O objetivo é que o desenvolvedor de um sistema distribuído faça uso de tais padrões para representar, em seu sistema, as funcionalidades escolhidas de uma plataforma de comunicação.

Estes padrões foram construídos e provados em Event-B e, assim, cada um define um passo de refinamento guiado no sistema alvo.

A Figura 3.1 apresenta uma visão geral dos padrões construídos. As linhas representam relações entre pares de especificação padrão e refinamento padrão, definindo cada uma um padrão. Nesta figura, no sentido vertical, de cima para baixo, tem-se redução de não determinismo: a especificação padrão está ao alto e o refinamento padrão abaixo, em cada padrão.

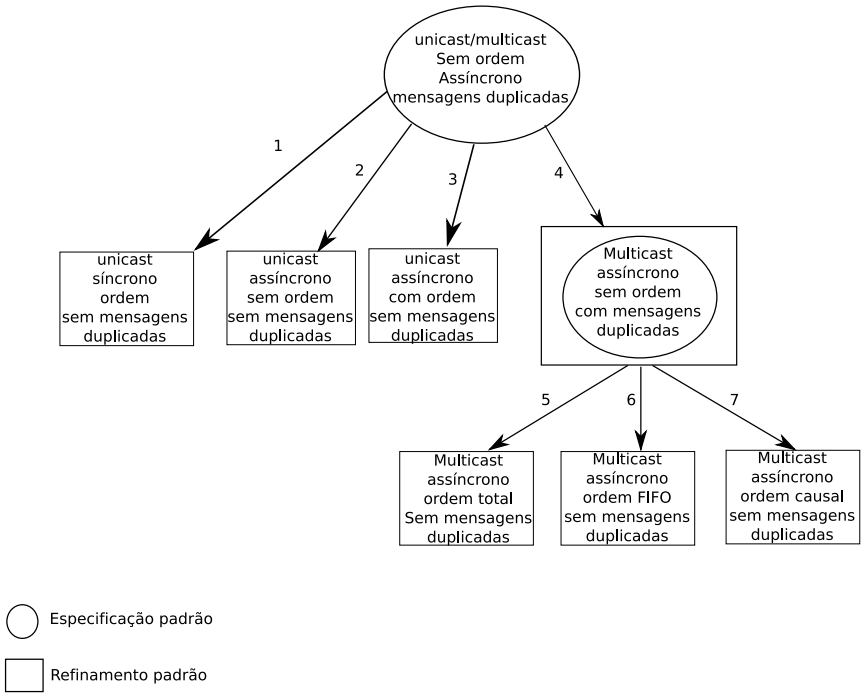


Figura 3.1 – Estrutura da biblioteca padrão

Nota-se que um refinamento padrão, em um padrão, pode conter os elementos da especificação padrão de um ou mais outros padrões, ensejando a possibilidade de planejar diversos passos de refinamento para chegar às funcionalidades desejadas. Por exemplo, veja na figura que a partir da aplicação do padrão 4 pode-se considerar a aplicação dos padrões 5, 6 ou 7.

Visando facilitar a aplicação dos padrões contidos na biblioteca para os mais diversos tipos de sistemas distribuídos modelados em Event-B, todas as definições de tipos de dados e constantes dos padrões foram construídas no contexto, parte estática do modelo. Com essa característica os padrões podem ser aplicados a diferentes tipos de dados, bastando alterações no respectivo contexto, não ficando restrita sua utilização a apenas uma construção.

Nas próximas seções serão descritos os padrões contidos na biblioteca de padrões. Os padrões existentes na biblioteca de padrões foram construídos aos pares: especificação padrão (modelo abstrato) e refinamento padrão (modelo concreto).

3.1 Especificação Padrão Inicial

O conjunto de padrões propostos, conforme Figura 3.1, tem a mesma especificação padrão de partida. O modelo especificação padrão inicial (círculo superior da Figura 3.1) define as estruturas básicas de comunicação necessárias para a troca de mensagens entre os processos. Para tanto, no contexto da especificação padrão de partida, apenas declarou-se o conjunto $DADOS$. Tal conjunto representa os tipo de dados que trafegaram no canal de comunicação do modelo.

O modelo da especificação padrão inicial, conforme Figura 3.2, contém três variáveis declaradas: $canaldados$, $dadosrecebidos$ e $msgenviadas$.

$$canaldados$$
$$dadosrecebidos$$
$$msgenviadas$$

Figura 3.2 – Variáveis da especificação padrão

A variável $canaldados$ representa o canal de comunicação entre o processo emissor e o processo receptor, sendo especificada como conjunto que contém uma função parcial de valores naturais (diferente de zero), mapeados para um valor do conjunto $DADOS$ (inv1 da Figura 3.3), ou seja, cada valor (mensagem) presente no $canaldados$ é uma associação entre uma identificação da mensagem a mensagem (dado). A função parcial ($\pfun$) permite nenhum ou infinitos valores mapeados.

INVARIANTS

inv1 : $canaldados \subseteq \mathbb{N} \pfun DADOS$

inv2 : $dadosrecebidos \subseteq \mathbb{N} \pfun DADOS$

inv3 : $msgenviadas \in \mathbb{N}_1$

Figura 3.3 – Invariantes da especificação padrão

A variável $dadosrecebidos$, representa no modelo o *buffer* do processo receptor, sendo especificada igualmente ao $canaldados$ (inv2 da Figura 3.3). Já a variavel $msgenviadas$, representa a identificação que é enviada junto do dado na mensagem, sendo especificada como um valor natural diferente de zero (inv3 da Figura 3.3).

O evento $initialisation$ da Figura 3.4 é reponsável por inicializar as variáveis do modelo. Inicialmente os conjuntos $canaldados$ e $dadosrecebidos$ são vazios (act1 e act2 da Figura 3.4) e variável $msgenviadas$ tem valor 1 (act3 da Figura 3.4).

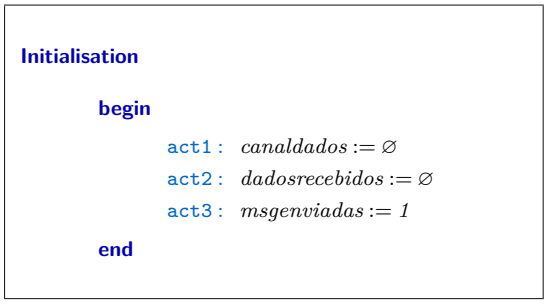


Figura 3.4 – Evento de inicialização da especificação padrão

Além do evento de inicialização, o modelo de especificação do padrão inicial, é constituído de dois eventos: $sender$ e $receiver$.

O evento $sender$ (Figura 3.5) é reponsável pelo envio de uma mensagem. Para tanto suas guardas deverão ser respeitadas, ou seja, a mensagem a ser enviada deve pertencer ao conjunto $DADOS$ (grd1 da Figura 3.5) e o identificador da mensagem deve ser um valor natural diferente de zero (grd2 da Figura 3.5). Sendo verdadeiras essas condições, o evento $sender$ coloca uma mensagem no canal de comunicação (act1 da Figura 3.5) e incrementa em uma unidade o identificador da mensagem (act2 da Figura 3.5) .

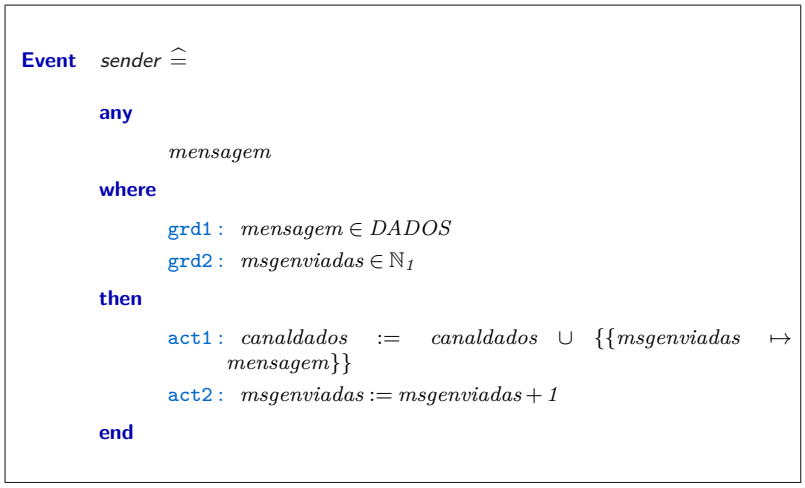


Figura 3.5 – Evento $sender$ da especificação padrão

Já o evento $receiver$ (Figura 3.6) é responsável por receber as mensagens que estão trafegando no canal. Portanto para este evento ocorrer, basta que exista uma mensagem no canal de comunicação (grd1 da Figura 3.6). Sendo verdadeira esta condição, a *buffer* recebe a mensagem do canal

de comunicação (act1 da Figura 3.6) e através de uma ação não determinística, a mensagem pode continuar ou ser removida do canal (act2 da Figura 3.6).

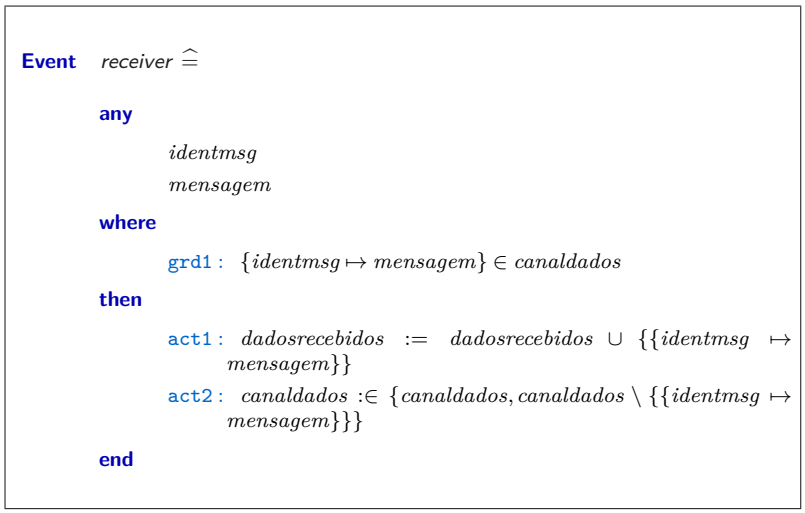


Figura 3.6 – Evento $receiver$ da especificação padrão

Dada a modelagem da especificação padrão inicial, nota-se que as estruturas previamente necessárias em um sistema alvo para aplicação do padrão são bastante simples. A modelagem do usuário se inicia com uma noção ainda pouco definida das características de comunicação desejadas.

Este modelo é ponto de partida para os vários padrões da biblioteca e define todas possíveis seqüências de computações, em termos de operações de envio e recepção de mensagens, que este canal possa apresentar. Os padrões descritos a seguir, restringem o conjunto possível de computações em passos de refinamento, adicionando estruturas, fortalecendo guardas e ações.

Ao escolher aplicar um padrão específico, o usuário opta por uma dada semântica de comunicação. Isso se concretiza com um passo de refinamento guiado que restringe não determinismo do sistema. Para um padrão da biblioteca proposta, o passo de refinamento definido elimina computações que não representem as características específicas do mecanismo de comunicação desejado.

3.2 Padrão *Unicast* Síncrono

O padrão *unicast* sícrono (1 da Figura 3.1), consiste em um mecanismo de comunicação para troca de mensagens entre um processo emissor e um processo receptor, sendo que uma operação de envio só tem sucesso com a respectiva operação de recepção.

Para controle das mensagens, enviadas pelo $sender$ para o $receiver$, no refinamento padrão do *unicast* síncrono, foram adicionados dois novos canais $canalsend$ e $canalrecv$, além de duas variáveis de controle $blocksend$ e $blockrecv$ (Figura 3.7).

Nas invariantes, definiu-se que as novas variáveis, adicionadas em um passo refinamento, somente poderão conter o valor 0 ou 1 no modelo (inv1, inv2, inv3 e inv4 da Figura 3.8). As invariantes inv6, inv7, inv8 da Figura 3.8 provam propriedades do sincronismo no sistema e serão discutidas adiante no trabalho.

Figura 3.7 – Variáveis do refinamento padrão do *unicast* síncrono

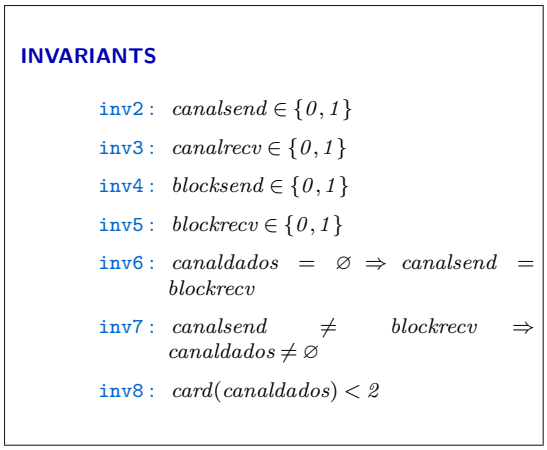


Figura 3.8 – Invariantes do refinamento padrão sincrono

No evento de inicialização do refinamento padrão, atribui-se o valor zero para as novas variáveis $canalsend$, $canalrecv$, $blocksend$ e $blockrecv$ (act5, act6, act7, act8 da Figura 3.9).

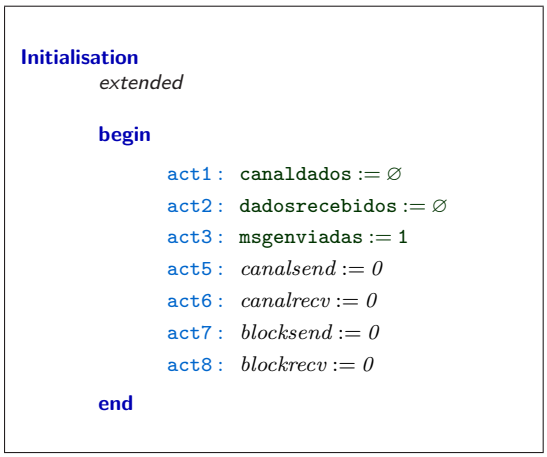


Figura 3.9 – Inicialização do refinamento padrão síncrono

Para garantir o sincronismo, foram adicionadas duas guardas no refinamendo do evento $sender$ (Figura 3.10), restrigindo o evento a enviar uma mensagem somente quando: receber a confirmação do $receiver$ (grd3 da Figura 3.10) e o $canaldados$ esteja vazio (grd4 da Figura 3.10). Além das guardas, foram adicionadas ações responsáveis por: avisar o $receiver$ da mensagem enviada (act3 da Figura 3.10) e atualizar o conhecimento local do $sender$ sobre a última mensagem enviada (act4 da Figura 3.10).

Já no refinamento do evento $receiver$, adicionou-se uma guarda restrigindo o evento para somente ocorrer quando o $sender$ identificar o envio de uma nova mensagem no $canalsend$ (gr2

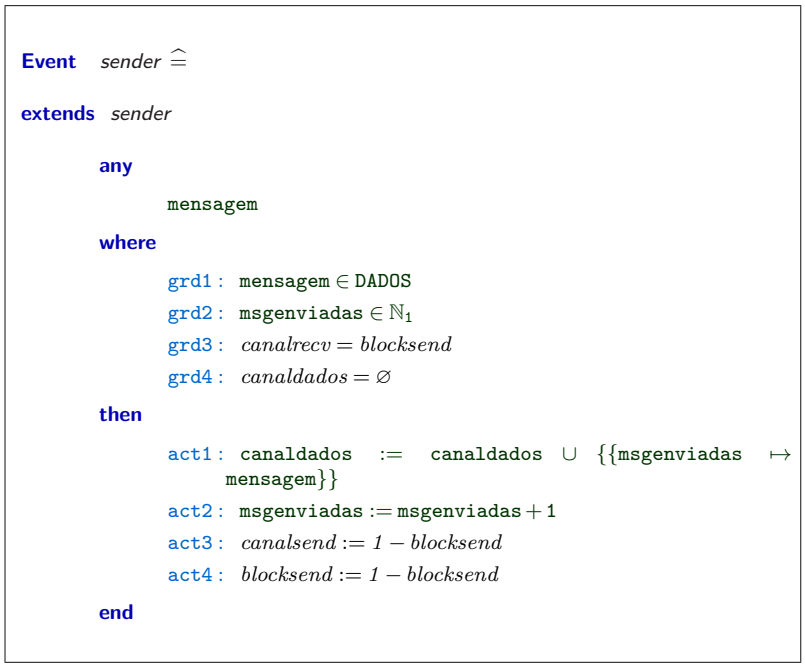


Figura 3.10 – Evento *sender* do refinamento padrão síncrono

da Figura 3.11). Nas ações do evento *receiver* removeu-se o não determinismo do modelo (act2 da Figura 3.10) e adicionou-se duas ações: uma para informar o *sender* sobre o recebimento da mensagem (act3 da Figura 3.10) e outra para atualizar o valor do *receiver* para a nova mensagem recebida.

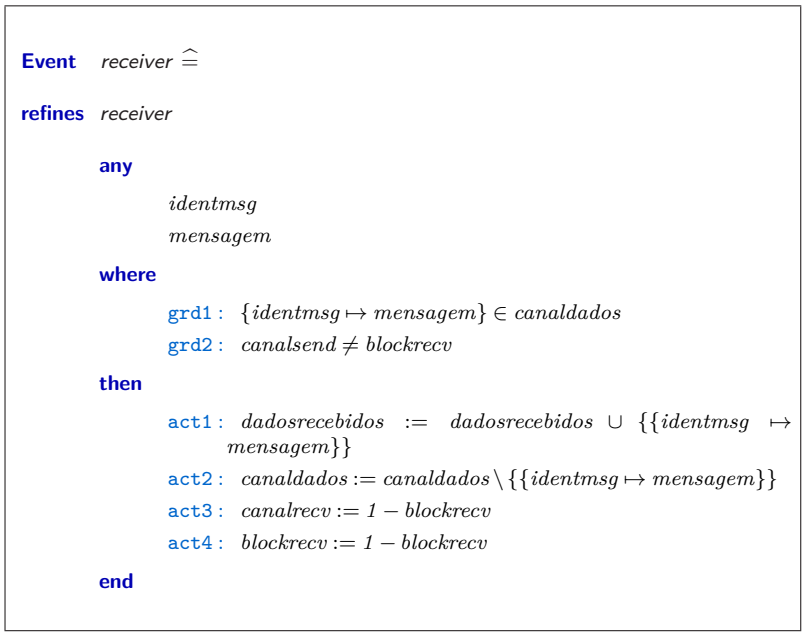


Figura 3.11 – Evento *receiver* do refinamento padrão síncrono

Dada a modelagem do refinamento padrão síncrono, nota-se que:

- os canais adicionados possuem a função de sincronia no modelo, ou seja, um é utilizado para avisar o evento *receiver* quando uma nova mensagem é colocada no canal, deixando o *sender*

bloqueado, e o outro para avisar o evento $sender$ quando a mensagem for recebida pelo evento $receiver$, portanto desbloqueando o sender.

- foi reduzido o não-determismo no evento $receiver$ fazendo que com as mensagens sejam removidas do canal após lidas pelo pelo receptor.
- a conservação de três invariantes contidas no modelo refinamento padrão, permite comprovar que no máximo uma mensagem por vez estará trafegando no canal (inv8 da Figura 3.8) e que enquanto a confirmação do $receiver$ não for concretizada o $sender$ permanecerá bloqueado (inv6 e inv7 da Figura 3.8).

3.3 Padrão *Unicast* Assíncrono sem Ordem

O padrão *unicast* assíncrono sem ordem (2 da Figura 3.1), consiste em um mecanismo de comunicação para troca de mensagens entre um processo emissor e um processo receptor. O processo emissor não fica bloqueado aguardando resposta, com isso um número ilimitado de mensagens podem estar trafegando no canal de comunicação. As mensagens são entregues em qualquer ordem.

No refinamento padrão as guardas dos eventos $sender$ e $receiver$ não foram modificadas, mantendo a mesma construção da especificação padrão. Apenas removeu-se o não determinismo na ação de recepção para que apenas uma cópia da mensagem seja recebida (act2 da Figura 3.12). Desta forma, não acontecerão recepções duplicadas.

Event *receiver* $\widehat{=}$

refines *receiver*

any

$identmsg$

$mensagem$

where

grd1 : $\{identmsg \mapsto mensagem\} \in canaldados$

then

act1 : $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$

act2 : $canaldados := canaldados \setminus \{\{identmsg \mapsto mensagem\}\}$

end

Figura 3.12 – Evento $receiver$ do refinamento padrão sem ordem

3.4 Padrão *Unicast* Assíncrono com Ordem.

O padrão *unicast* assíncrono com ordem (3 da Figura 3.1), consiste em um mecanismo de comunicação, para troca de mensagens entre um processo emissor e um processo receptor, com ordenação FIFO. Novamente, um número ilimitado de mensagens podem estar no canal. Entretanto,

a modelagem realizada no refinamento do padrão *unicast* assíncrono com ordem é diferenciada das anteriores. Neste, foi adicionado uma variável $ordemrecv$ (Figura 3.13) ao refinamento e restringido o evento $receiver$ para respeitar a entrega das mensagens na ordenação FIFO.

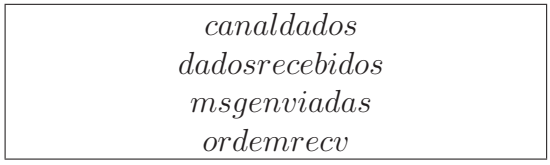


Figura 3.13 – Variáveis do refinamento padrão do *unicast* ordenado

A variável $ordemrecv$ representa a ordem de recepção das mensagens no evento $receiver$, para tanto foi definada pela invariante do modelo como um valor natural diferente de zero (inv1 da Figura 3.14).

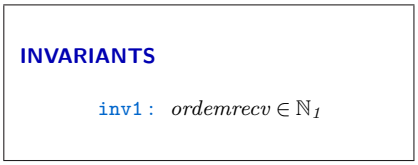


Figura 3.14 – Invariantes do refinamento padrão ordenado

No evento de inicialização do refinamento padrão, foi adicionada a inicialização da variável $ordemrecv$. Na Figura 3.15 é possivel perceber que $ordemrecv$ é inicializada com valor 1 (act4).

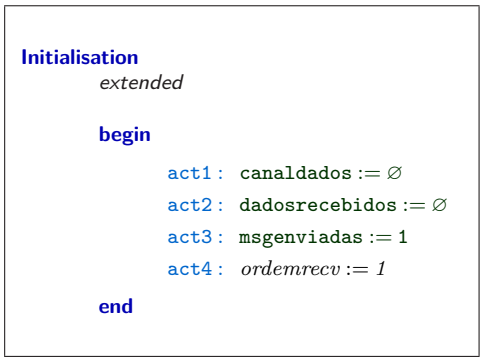


Figura 3.15 – Invariantes do refinamento padrão ordenado

Neste refinamento, não foram realizadas modificações no evento $sender$ da especificação padrão (Seção 3.1). A variável $msgenviadas$, do modelo abstrato (especificação padrão da Seção 3.1), neste refinamento, representa um identificador de ordem global adicionado a cada nova mensagem no canal de comunicação entre o $sender$ e o $receiver$.

No evento $receiver$, foram adicionadas: uma guarda (grd2 da Figura 3.16) para restringir a entrega das mensagens de acordo com a ordem da variável $ordemrecv$ e uma ação para incrementar, na ocorrencia de uma recepção, a variável $ordemrecv$ em uma unidade (act3 da Figura 3.16).

Além disso, após cada recepção, o evento $receiver$ remove a mensagem do $canaldados$ (act1 da Figura 3.16).

Dada a modelagem do refinamento padrão do *unicast* ordenado, nota-se que:

Event *receiver* $\widehat{=}$

refines *receiver*

any

$identmsg$

$mensagem$

where

grd1 : $\{identmsg \mapsto mensagem\} \in canaldados$

grd2 : $identmsg = ordemrecv$

then

act1 : $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$

act2 : $canaldados := canaldados \setminus \{\{identmsg \mapsto mensagem\}\}$

act3 : $ordemrecv := ordemrecv + 1$

end

Figura 3.16 – Evento $receiver$ do refinamento padrão ordenado

- com a utilização de uma ordenação global, associada ao envio de cada mensagens, e com a existência de uma guarda, que indica a próxima mensagem a ser recebida pelo evento $receiver$ ($identmsg = ordemrecv$), foi possível garantir a ordenação FIFO de entrega da mensagens.
- com a existencia da ordenação e com a remoção da mensagem após cada recepção, removeu-se o não determinismo existente na espeficicação padrão do modelo.

3.5 Padrão *Multicast* Assíncrono sem Ordem

O padrão *multicast* assíncrono sem ordem (4 da Figura 3.1), consiste em um mecanismo de comunicação em grupo onde todo processo deste pode ser enviador ou receptor.

Dada tal característica, tornam-se necessária estruturas no modelo, de modo a identificar os diferentes processos participantes e o grupo *multicast* dos mesmos. Para tanto, no contexto deste refinamento, foram adicionadas duas constantes: NP que representa o último processo do modelo e $GRUPO$ definido como um conjunto que contém os processos participantes do grupo *multicast*. No modelo refinamento padrão, foi adicionada uma variável $canalprocessos$ (Figura 3.17) que contém os dados de origem e destino das mensagens *multicast*.

$canaldados$
$dadosrecebidos$
$msgenviadas$
$canalprocessos$

Figura 3.17 – Variáveis do refinamento padrão *multicast* sem ordem

A variável $canalprocessos$ é definida como um conjunto que contém: uma função parcial de um valor natural, diferente de zero, mapeado para um função parcial de um intervalo de 1 até NP, mapeada para o *power set* (todas possíveis combinações) do mesmo intervalo (Figura 3.18).

Portanto no $canalprocessos$, os processos originadores são representados por um intervalo de 1 até NP ($1 \mathinner{..} NP$ da inv1 da Figura 3.18) e o grupo *multicast* pelo *power set* deste mesmo intervalo ($\mathbb{P}(1 \mathinner{..} NP)$ da inv1 da Figura 3.18).

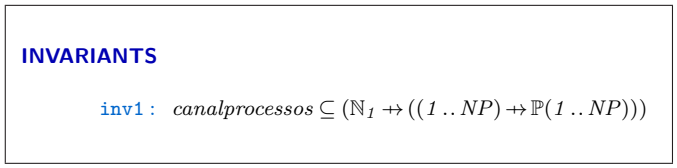


Figura 3.18 – Invariantes do refinamento padrão *multicast* sem ordem

O novo canal adicionado foi inicializado com vazio no evento de inicialização da máquina (act4 da Figura 3.19).

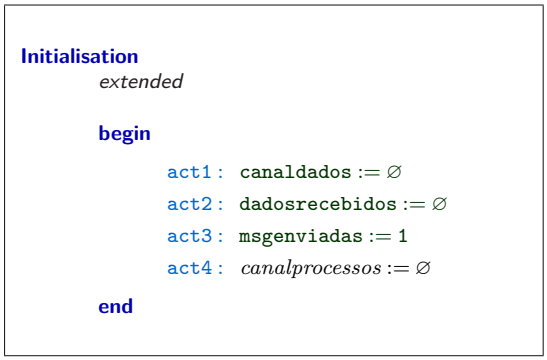


Figura 3.19 – Evento de inicialização do refinamento padrão do multicast sem ordem

No evento $sender$ foram adicionadas uma guarda e uma ação. A guarda é responsável por identificar um processo válido, como originador da mensagem (grd3 da Figura 3.20) e a ação por adicionar no $canalprocessos$, para cada mensagem postada no canal de comunicação ($canaldados$), a identificação da mensagem, o originador e o grupo de destinatários.

Analisando as ações do evento $sender$, é possivel perceber que o $canalprocessos$ é relacionado com o canal de mensagens ($canaldados$) através de uma chave de identificação única que é a ordenação global de envio das mensagens ($msgenviadas$).

No evento $receiver$ foram adicionadas duas guardas e uma ação. Existindo um grupo de processos como destinatário para uma mensagem ($grupotemp$ da Figura 3.21), as guardas acrescentadas no refinamento padrão informam um possível processo receptor (grd2 e grd3 da Figura 3.21). Já a ação adicionada, representa a remoção ou não da mensagem no canal $canalprocessos$, ou seja, representa um não determinismo sobre o canal com os dados de origem e destino da mensagem (act3 da Figura 3.21).

Dada a modelagem do refinamento padrão do *multicast* sem ordem, nota-se que:

- neste passo de refinamento foram introduzidos estruturas nos eventos, de modo a proporcionar a recepção e o envio de mensagens por múltiplos processos.

Event *sender* $\widehat{=}$

extends *sender*

any

$\texttt{mensagem}$

$processoenvia$

where

$\texttt{grd1}:\ \texttt{mensagem} \in \texttt{DADOS}$

$\texttt{grd2}:\ \texttt{msgenviadas} \in \mathbb{N}_1$

$\texttt{grd3}:\ processoenvia \in 1\,..\,NP$

then

$\texttt{act1}:\ \texttt{canaldados} := \texttt{canaldados} \cup \{\{\texttt{msgenviadas} \mapsto \texttt{mensagem}\}\}$

$\texttt{act2}:\ \texttt{msgenviadas} := \texttt{msgenviadas} + 1$

$\texttt{act3}:\ canalprocessos := canalprocessos \cup \{\{msgenviadas \mapsto \{(processoenvia \mapsto GRUPO)\}\}\}$

end

Figura 3.20 – Evento $sender$ do refinamento padrão do multicast sem ordem

Event *receiver* $\widehat{=}$

refines *receiver*

any

$identmsg$

$mensagem$

$processo$

$grupotemp$

$processorecv$

where

$\texttt{grd1}:\ \{identmsg \mapsto mensagem\} \in canaldados$

$\texttt{grd2}:\ \{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$

$\texttt{grd3}:\ processorecv \in grupotemp$

then

$\texttt{act1}:\ dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$

$\texttt{act2}:\ canaldados :\in \{canaldados, canaldados \setminus \{\{identmsg \mapsto mensagem\}\}\}$

$\texttt{act3}:\ canalprocessos :\in \{canalprocessos, canalprocessos \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}\}$

end

Figura 3.21 – Evento $receiver$ do refinamento padrão do multicast sem ordem

- $canalprocessos$ é relacionado com o canal de mensagens ($canaldados$) através de uma chave de identificação única, ou seja, para cada mensagem postada no canal de comunicação ($canaldados$) é especificado o originador e o grupo de destinatários (estruturas de comunicação *multicast*).

- os diferentes processos pertencentes ao grupo *multicast* podem receber as mensagens do canal nas mais diferentes ordens, assim como também podem postergar o recebimento de determinada mensagem ou receber duplicatas.

- existe a presença de não determinísmo sobre os canais de comunicação do modelo.

- este refinamento padrão define todas possíveis ordens de entrega para o grupo e, portanto, serve como especificação padrão para os outros padrões *multicast* (ordem FIFO, total e causal) (Figura 3.1).

3.6 Padrão *Multicast* FIFO

O padrão *multicast* assíncrono com ordem FIFO (6 da Figura 3.1), consiste em um mecanismo de comunicação em grupo com entrega FIFO. No refinamento padrão, do *multicast* FIFO, foram inseridas novas variáveis ao modelo, sendo as mesmas apresentadas na Figura 3.22.

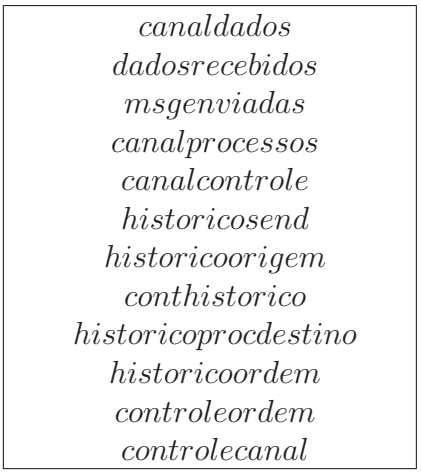


Figura 3.22 – Variáveis do refinamento padrão *multicast* FIFO

A variável $canalcontrole$ representa o canal de comunicação que identifica a ordem FIFO das mensagens postadas, sendo definida como uma função parcial de naturais, sem o zero, mapeada para naturais (inv1 da Figura 3.23).

A ordem da última mensagem enviada, por cada processo emissor, é armazenada no $historicosend$. No refinamento padrão, $historicosend$ é definida como uma função total dos processos do modelo, mapeada para um valor natural (inv2 da Figura 3.23). Uma função total determina que todo elemento, do domínio, tem que possuír um e somente um elemento, da imagem, mapeado.

As variáveis $historicoorigem$, $historicoprocdestino$ e $historicoordem$ representam o histórico das mensagens recebidas pelos processos. Portanto, registrando após cada recepção o processo de origem, o processo de destino e a ordem FIFO da mensagem. Estes três históricos são definidos respectivamente pelas invariantes inv3, inv5 e inv6 da Figura 3.23.

Os históricos do modelo ($historicoorigem$, $historicoprocdestino$ e $historicoordem$) são associados pela variável $conthistorico$, que somente pode conter valores naturais (inv4 da Figura 3.23). Já as variáveis $controleordem$ e $controlecanal$, são variáveis que armazenam respectivamente: a ordem da última mensagem recebida por cada processo e o total de recepções realizado em cada mensagem. As invariantes inv7 e inv8 da Figura 3.23 são responsáveis por definir as variáveis

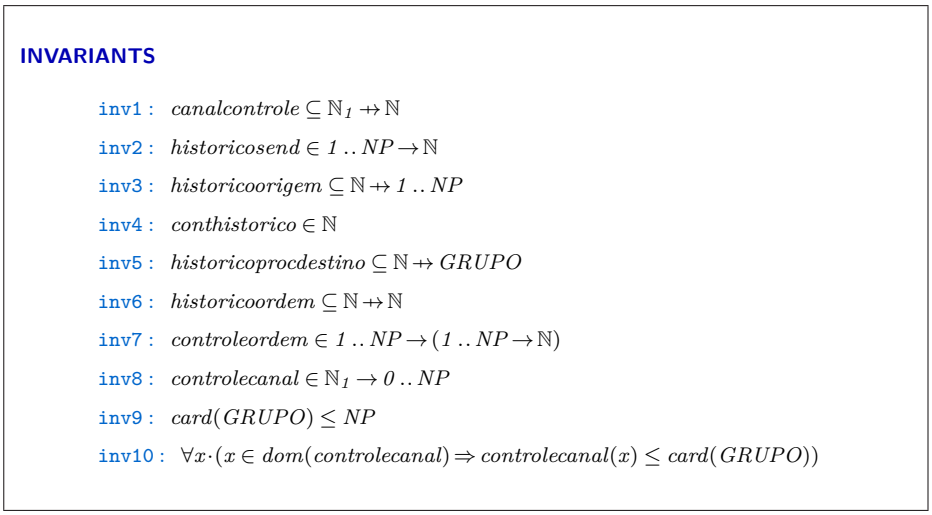


Figura 3.23 – Invariantes do refinamento padrão *multicast* FIFO

$controleordem$ e $controlecanal$. As invariantes inv9 e inv10 provam propriedades sobre o padrão, portanto serão discutidas adiante no texto.

O evento de inicialização das variáveis do refinamento, pode ser visualizado na Figura 3.24. Assim como os canais herdados do modelo abstrato (especificação padrão), o $canalcontrole$ é inicializado vazio (act5 da Figura 3.24). A ordem FIFO ($historicosend$) de todos os processos emissores do modelo (1 até NP), é mapeada para o valor zero (act6 3.24). Os historicos do modelo são inicializados vazios (act7, act9 e act10 da Figura 3.24).

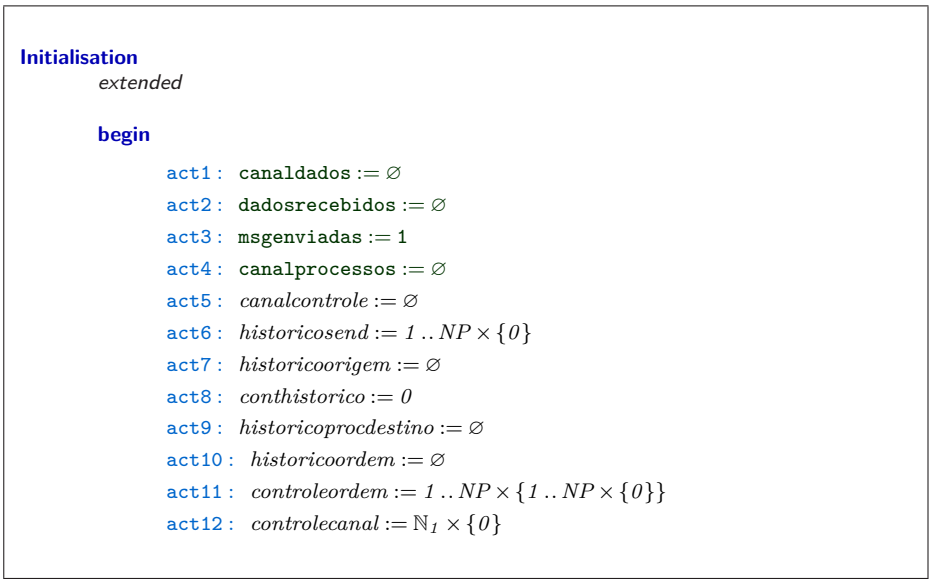


Figura 3.24 – Evento de inicialização das variáveis do modelo

O indexador dos historicos é iniciado com zero (act8 da Figura 3.24). A ordem FIFO de recepção dos processos é mapeada para o valor zero na inicialização (act11 da Figura 3.24). A variável $controlecanal$ tem todos os seus elemento mapeados para zero.

Além do evento de inicialização, são apresentados outros três eventos no refinamento padrão: o $sender$ e o $receiver$ herdado da especificação padrão (Seção 3.5) e o evento $removemsgcanal$

adicionado neste passo de refinamento (refinamento padrão).

No $sender$, do refinamento padrão, foram adicionadas duas ações: uma reponsável por adicionar no $canalcontrole$ a ordem FIFO da mensagem postada (act4 da Figura 3.25), sendo relacionada com os outros canais por um identificador único ($msgenviadas$) e outra ação responsável por incrementar em uma unidade o $historicosend$ (act5 da Figura 3.25),ou seja, atualizando a ordem FIFO do processo emissor.

Event *sender* $\widehat{=}$

refines *sender*

any

$mensagem$
$processoenvia$

where

grd1 : $mensagem \in DADOS$
grd2 : $msgenviadas \in \mathbb{N}_1$
grd3 : $processoenvia \in 1 \mathinner{..} NP$

then

act1 : $canaldados := canaldados \cup \{\{msgenviadas \mapsto mensagem\}\}$
act2 : $msgenviadas := msgenviadas + 1$
act3 : $canalprocessos := canalprocessos \cup \{\{msgenviadas \mapsto \{(processoenvia \mapsto GRUPO)\}\}\}$
act4 : $canalcontrole := canalcontrole \cup \{\{msgenviadas \mapsto (historicosend(processoenvia) + 1)\}\}$
act5 : $historicosend(processoenvia) := historicosend(processoenvia) + 1$

end

Figura 3.25 – Evento $sender$ do refinamento padrão *multicast* FIFO

A recepção das mensagens, no refinamento, é divida em dois eventos: $receiver$ e $removemsgcanal$. A última recepção de cada mensagem é sempre realizada pelo evento $removemsgcanal$, de modo que seja realizada a recepção e a remoção da mensagem do canal. Os outros casos de recepção acontecem no evento $receiver$.

As guardas do evento $receiver$ podem ser visualizadas na Figura 3.26. As guardas grd8 e grd9 da Figura 3.26 são responsáveis por identificar a ordem FIFO da última mensagem recebida do processo originador para o processo receptor. Já as guardas grd1, grd2 e grd3 da Figura 3.26 são herdadas da especificação padrão (Seção 3.5).

Além das guardas citadas anteriomente, para o evento $receiver$ ocorrer, as seguintes guardas devem ser verdadeiras: deve existir uma dado, com a ordem FIFO da mensagem, no $canalcontrole$ (grd4 da Figura 3.26); o processo originador deve ser um processo válido do modelo (grd5 da Figura 3.26); o processo receptor deve pertencer ao grupo *multicast* (grd6 da Figura 3.26); a ordem FIFO da mensagem deve ser um valor natural diferente de zero (grd7 da Figura 3.26);levando em conta o mesmo processo de origem e destino (grd2, grd3, grd4, grd8 e grd9 da Figura 3.26), a ordem da mensagem do canal de comunicação deverá ser uma unidade maior que a ordem da última recepção (grd10 da Figura 3.26); o identificador usado para associar os canais deve pertencer ao domínio do $controlecanal$ (grd11 da Figura 3.26); a mensagem em questão não pode ter sido recebida por

todos os processos do grupo *multicast* (grd12 da Figura 3.26); o processo receptor deve pertencer ao grupo de processos válidos (grd13 da Figura 3.26); as ordens FIFO do processo receptor devem pertencer a imagem de $controleordem$ (grd14 da Figura 3.26).

grd1 $\{identmsg \mapsto mensagem\} \in canaldados$
grd2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$
grd3 $processorecv \in grupotemp$
grd4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole$
grd5 $processo \in 1..NP$
grd6 $processorecv \in GRUPO$
grd7 $ordemprocesso \in \mathbb{N}_1$
grd8 $processorecv \mapsto conjuntotemp \in controleordem$
grd9 $processo \mapsto ordemtemp \in conjuntotemp$
grd10 $ordemprocesso = ordemtemp + 1$
grd11 $identmsg \in dom(controlecanal)$
grd12 $controlecanal(identmsg) < (card(GRUPO) - 1) \wedge controlecanal(identmsg) < NP$
grd13 $processorecv \in 1..NP$
grd14 $conjuntotemp \in ran(controleordem)$

Figura 3.26 – guardas do evento $receiver$ do refinamento padrão *multicast* FIFO

Sendo verdeiras as guardas do evento $receiver$, são realizadas as seguintes ações nas variáveis do modelo: a *buffer* recebe a mensagem enviada (act1 da Figura 3.27); a mensagem recebida pelo processo continua presente no canal de comunicação (act2 da Figura 3.27); os historicos $historicoorigem$, $historicoprocdestino$ e $historicoordem$ são atualizados com os novos valores da recepção (act3, act5 e act6 da Figura 3.27); levando-se em conta o processo originador e o processo receptor da mensagem recebida, atualizada-se a ordem FIFO da variável de controle, para o valor contido na mensagem (act7 da Figura 3.27); para que os outros processos do grupo *multicast* possam receber a mensagem, $canalprocessos$ continua com a mensagem no canal (act8 da Figura 3.27); $controlecanal$ e $conthistorico$ são atualizados em uma unidade (respectivamente act9 e act4 da Figura 3.27).

act1 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act2 $canaldados := canaldados$
act3 $historicoorigem := historicoorigem \cup \{\{conthistorico \mapsto processo\}\}$
act4 $conthistorico := conthistorico + 1$
act5 $historicoprocdestino := historicoprocdestino \cup \{\{conthistorico \mapsto processorecv\}\}$
act6 $historicoordem := historicoordem \cup \{\{conthistorico \mapsto ordemprocesso\}\}$
act7 $controleordem := controleordem \Leftarrow \{processorecv \mapsto (conjuntotemp \Leftarrow \{processo \mapsto ordemprocesso\})\}$
act8 $canalprocessos := canalprocessos$
act9 $controlecanal(identmsg) := controlecanal(identmsg) + 1$

Figura 3.27 – ações do evento $receiver$ do refinamento padrão *multicast* FIFO

Na Figura 3.28 são descrita as guardas do evento $removemsgcanal$. Comparando-se as guardas do evento $receiver$ (Figura 3.26), com as guardas do evento $removemsgcanal$ (Figura 3.28) é possivel perceber que, com exceção da grd12, as guardas que restrigem os dois eventos são as mesmas. Assim como as guardas, as ações dos eventos $removemsgcanal$ (Figura 3.27) e $receiver$ (Figura 3.29) são iguais, com exceção das ações act2, act3 e act4.

As semelhanças encontradas nas guardas e nas ações dos eventos $receiver$ e $removemsgcanal$, deve-se ao fato que, as restrições para recepção das mensagens são necessárias a ambos eventos. Entretanto como $removemsgcanal$, além da recepção, é responsável pela remoção da mensagem

grd1 $\{identmsg \mapsto mensagem\} \in canaldados$
grd2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$
grd3 $processorecv \in grupotemp$
grd4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole$
grd5 $processo \in 1 \mathinner{..} NP$
grd6 $processorecv \in GRUPO$
grd7 $ordemprocesso \in \mathbb{N}_1$
grd8 $processorecv \mapsto conjuntotemp \in controleordem$
grd9 $processo \mapsto ordemtemp \in conjuntotemp$
grd10 $ordemprocesso = ordemtemp + 1$
grd11 $identmsg \in dom(controlecanal)$
grd12 $controlecanal(identmsg) = (card(GRUPO) - 1)$
grd13 $conjuntotemp \in ran(controleordem)$
grd14 $processorecv \in 1 \mathinner{..} NP$

Figura 3.28 – guardas do evento $removemsgcanal$ do refinamento padrão *multicast* FIFO

do canal, algumas de suas ações e guardas tiveram que ser modificadas para preservar tal comportamento. Para tanto especificou-se, a guarda grd12 do evento $removemsgcanal$, para que o evento somente ocorra quando as mensagens do canal, possuirem apenas um processo como destino. As ações act2, act3 act4 da Figura 3.29 foram modeladas para remover a mensagem do $canaldados$ e as informações associadas a mensagem, existentes nos outros canais.

act1 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act2 $canaldados := canaldados \setminus \{\{identmsg \mapsto mensagem\}\}$
act3 $canalcontrole := canalcontrole \setminus \{\{identmsg \mapsto ordemprocesso\}\}$
act4 $canalprocessos := canalprocessos \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}$
act5 $conthistorico := conthistorico + 1$
act6 $historicoorigem := historicoorigem \cup \{\{conthistorico \mapsto processo\}\}$
act7 $historicoprocdestino := historicoprocdestino \cup \{\{conthistorico \mapsto processorecv\}\}$
act8 $historicoordem := historicoordem \cup \{\{conthistorico \mapsto ordemprocesso\}\}$
act9 $controleordem := controleordem \Leftarrow \{processorecv \mapsto (conjuntotemp \Leftarrow \{processo \mapsto ordemprocesso\})\}$
act10 $controlecanal(identmsg) := controlecanal(identmsg) + 1$

Figura 3.29 – guardas do evento $removemsgcanal$ do refinamento padrão *multicast* FIFO

Dada a modelagem do refinamento padrão do *multicast* FIFO, nota-se que:

- foi definido um canal adicional contendo a ordenação FIFO das mensagens enviadas pelos processos. Desta forma, o refinamento ficou constituído de três canais de comunicação: o canal com as mensagens (herdado da especificação padrão inicial 3.1); o canal com as informações de originador e destinatário (herdado do refinamento padrao *multicast* assíncrono sem ordem 3.5); e, por fim, o canal com informação acerca da ordem FIFO, introduzida neste passo de refinamento.

- para realizar a ordenção FIFO, modelou-se uma estrutura que armazena as mensagens recebidas ($controleordem$), de cada processo originador e destinatário.

- com base nas restrições, adicionadas neste passo de refinamento, os eventos responsáveis pelas recepções de mensagens, verificam qual a próxima possibilidade de recepção para cada processo do grupo *multicast*, ou seja, para ser realizada a recepção de uma mensagem, a ordem do processo de origem, associada a mensagem, deve ser uma unidade maior que a existente no processo receptor (grd10 das Figuras 3.26 e 3.28).

- foi adicionado um evento, que na última recepção de cada mensagem, remove a mensagem do canal de comunicação e as informações associadas a ela. Desta forma, removendo o não determinísmo sobre os canais, ou seja, restringindo o comportamento do modelo.
- na inv9 da Figura 3.23 foi possivel provar que o grupo *multicast* terá o máximo NP elementos. Já a inv10 define que o número máximo de recepções será o número de elementos do grupo, poratnto nenhum processo realizará a recepção de uma mesma mensagem duas vezes.

3.7 Padrão *Multicast* Total

O padrão *multicast* assíncrono com ordem total (5 da Figura 3.1), consiste em um mecanismo de comunicação em grupo onde todos processos recebem as mensagens na mesma ordem global do seu envio.

No refinamento padrão do multicast total, foram inseridas duas novas variáveis: $ordem$ e $historicoprocesso$ (Figura 3.30).

$$
\begin{array}{c}
canaldados \\
dadosrecebidos \\
msgenviadas \\
canalprocessos \\
ordem \\
historicoprocesso
\end{array}
$$

Figura 3.30 – variáveis do refinamento padrão do *multicast* total

A variável $ordem$ é utilizada na remoção das mensagens do canal de comunicação, sendo definida pela inv1 da Figura 3.31 como um valor natural. Já a variável $historicoprocesso$ é responsável por registrar a ordem das recepções das mensagens pelos processos, ou seja, representa a ordem total no modelo. $Historicoprocesso$ é definida, pela inv2 da Figura 3.31, como um elemento de uma função total de 1 até NP mapeado para valores naturais.

INVARIANTS

inv1 : $ordem \in \mathbb{N}$

inv2 : $historicoprocesso \in 1 \mathinner{..} NP \rightarrow \mathbb{N}$

Figura 3.31 – Invariantes do refinamento padrão *multicast* FIFO

O refinamento padrão, do padrão *multicast* total, contém quatro eventos: o evento de inicialização, o evento $sender$, o evento receiver e por último o evento $removemsgcanal$, adicionado neste passo de refinamento.

No evento de incialização, foram adicionadas duas novas ações: $ordem$ é iniciada com valor 1 (act5 da Figura 3.32) e $historicoprocessos$ tem os valores de 1 até NP mapeados para zero (act7 da Figura 3.32).

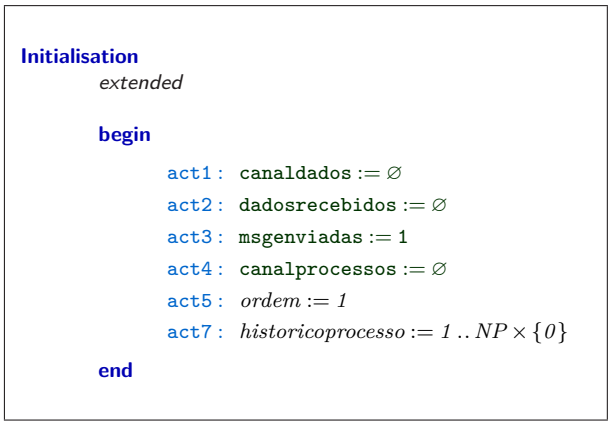


Figura 3.32 – Evento de inicialização do refinamento padrão *multicast* total

No refinamento da especificação padrão, não foi realizada nenhuma modificação no evento $sender$. Portanto, no refinamento padrão, do padrão *multicast* total, o evento $sender$ continua igual ao descrito na Seção 3.5.

Igualmente ao padrão *multicast* FIFO, a recepção das mensagens, no padrão *multicast* total, foi divida em dois eventos: $receiver$ e $removemsgcanal$.

No evento $receiver$, a recepção de uma dada mensagem ocorre até o penúltimo processo do grupo *multicast*. Para tanto, as guardas do evento $receiver$ deverão ser verdadeiras: deve existir uma mensagem no $canaldados$ (grd1 da Figura 3.33) e um dado associado ($identmsg$) no $canalprocessos$ com a origem e o destino da mensagem (grd2 da Figura 3.33); a recepção deve ser realizada por um processo pertencer ao grupo de destino da mensagem (grd3 da Figura 3.33); a ordem total de envio das mensagens deve ser respeitada na recepção (grd4 da Figura 3.33); a recepção deve ser realizada até o penúltimo processo do grupo *multicast* (grd5 da Figura 3.33).

Sendo verdadeiras as guardas do evento $receiver$, as seguintes ações são realizadas: a *buffer* recebe a mensagem (act1 da Figura 3.33);visando futuras recepções, por outros processos do grupo *multicast*, a mensagem continua no canal de comunicação (act2 da Figura 3.33); o historico de recepção do processo é atualizado para ordem contida na mensagem (act3 da Figura 3.33); pelo mesmo motivo do canal de comunicação ($canaldados$), o dado associado a mensagem é mantido no canal (act4 da Figura 3.33).

O evento $removemsgcanal$ é responsável pela recepção das mensagens, entretanto, neste evento, somente é realizada a recepção quando restar apenas um processo para receber determinada mensagem. Além de realizar a recepção, o evento $removemsgcanal$ deve remover a mensagem do canal de comunicação.

Para o evento $removemsgcanal$ ocorrer, suas guardas deverão ser verdadeiras: deve exitir uma mensagem no canal de comunicação (grd1 da Figura 3.34) e uma dado associado com a origem e o destino da mensagem (grd2 da Figura 3.34); o processo receptor deve pertencer ao grupo de destino da mensagem (grd3 da Figura 3.34); deve apenas faltar um processo para realizar a recepção da mensagem em questão (grd5 da Figura 3.34); A ordem total das mensagens enviadas deve ser preservada (grd7 da Figura 3.34).

Event *receiver* $\widehat{=}$

refines *receiver*

any

$identmsg$
$mensagem$
$processo$
$grupotemp$
$processorecv$

where

grd1 : $\{identmsg \mapsto mensagem\} \in canaldados$
grd2 : $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$
grd3 : $processorecv \in grupotemp$
grd4 : $processorecv \in dom(historicoprocesso) \wedge (identmsg = (historicoprocesso(processorecv) + 1))$
grd5 : $\neg(\forall proc \cdot (proc \neq processorecv \wedge proc \in dom(historicoprocesso) \Rightarrow historicoprocesso(proc) \geq ordem))$

then

act1 : $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act2 : $canaldados := canaldados$
act3 : $historicoprocesso(processorecv) := identmsg$
act4 : $canalprocessos := canalprocessos$

end

Figura 3.33 – Evento de $receiver$ do refinamento padrão *multicast* total

Sendo verdadeiras estas condições, são aplicadas as seguintes ações no modelo: a *buffer* recebe a mensagem presente no canal de comunicação (act1 da Figura 3.34); a mensagem é removida do canal de comunicação (act2 da Figura 3.34); a variável de controle da ordem total é incrementada em uma unidade (act3 da Figura 3.34); os dados associados a mensagem do canal de comunicação são removidos do canal (act4 da Figura 3.34); o histórico de recepção das mensagens recebe a ordem contida na mensagem recebida (act5 da Figura 3.34).

Dada a modelagem do padrão *multicast* total, nota-se que:

- no modelo do refinamento padrão do *multicast* total não foi inserida nenhuma estrutura nova de canal. Os canais presentes no modelo são herdados da especificação padrão (Seção 3.5).

- para a ordenação total das mensagens, foi utilizada a chave de identificação única, introduzida no padrão *multicast* sem ordem. Desta forma, a ordenação total, foi garantida através restrição das guardas dos eventos receptores, ou seja, um processo receptor somente receberá a mensagem que contiver uma ordem com o valor de uma unidade maior da contida no seu histórico (grd4 da Figura 3.33 e grd7 da Figura 3.34).

- foi introduzido um novo evento para remoção das mensagens dos canal de comunicação, ou seja, removeu-se o não determinísmo do modelo existente na especificação padrão.

Event *removemsgcanal* $\widehat{=}$

refines *receiver*

any

$identmsg$
$mensagem$
$processo$
$grupotemp$
$processorecv$

where

grd1 : $\{identmsg \mapsto mensagem\} \in canaldados$
grd2 : $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$
grd3 : $processorecv \in grupotemp$
grd5 : $identmsg = ordem$
grd6 : $\forall proc \cdot (proc \neq processorecv \wedge proc \in dom(historicoprocesso) \Rightarrow historicoprocesso(proc) \geq ordem)$
grd7 : $processorecv \in dom(historicoprocesso) \wedge (identmsg = (historicoprocesso(processorecv) + 1))$

then

act1 : $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act2 : $canaldados := canaldados \setminus \{\{identmsg \mapsto mensagem\}\}$
act3 : $ordem := ordem + 1$
act4 : $canalprocessos := canalprocessos \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}$
act5 : $historicoprocesso(processorecv) := identmsg$

end

Figura 3.34 – Evento de $removemsgcanal$ do refinamento padrão *multicast* total

3.8 Padrão *Multicast* Causal

O padrão *multicast* assíncrono com ordem causal (7 da Figura 3.1), consiste em um mecanismo de comunicação em grupo onde a recepção de mensagens respeita a relação de causa.

A modelagem da ordenação causal, descrita neste refinamento, é baseado no conceito de relógios lógicos descritos em [Lam78].

As variáveis presente no refinamento padrão, do padrão *multicast* causal, podem ser visualizadas na Figura 3.35.

$canaldados$
$dadosrecebidos$
$msgenviadas$
$canalprocessos$
$historicoorigem$
$conthistorico$
$historicoprocdestino$
$historicoordem$
$vetorclocks$
$canalordem$
$controlecanal$

Figura 3.35 – Variáveis do padrão *multicast* ordem causal

As variáveis inseridas neste passo de refinamento são: $historicoorigem$, $historicoprocdestino$, $historicoordem$, $conthistorico$, $vetorclocks$, $canalordem$ e $controlecanal$.

As variáveis $historicoorigem$, $historicoprocdestino$ e $historicoordem$ representam o histórico de recepção das mensagens pelos processos do grupo *multicast* e são associadas pela variável $conthistorico$. $Historicoorigem$ foi definida como um conjunto que deve conter funções parcias de números naturais mapeados para valores de 1 até NP (inv3 da Figura 3.36). A variável $historicoprocdestino$ foi definida como um conjunto que deve conter funções parcias de números naturais, mapeados para valores de GRUPO (inv5 da Figura 3.36). Já a variável $historicoordem$, foi definida como um conjunto que contém funções parcias de números naturais, mapeados para número naturais (inv6 da Figura 3.36). $Conthistorico$ foi definido como um valor natural (inv4 da Figura 3.36).

INVARIANTS

inv3 : $historicoorigem \subseteq \mathbb{N} \pfun 1 \upto NP$

inv4 : $conthistorico \in \mathbb{N}$

inv5 : $historicoprocdestino \subseteq \mathbb{N} \pfun GRUPO$

inv6 : $historicoordem \subseteq \mathbb{N} \pfun \mathbb{N}$

inv7 : $vetorclocks \in 1 \upto NP \fun (1 \upto NP \fun \mathbb{N})$

inv8 : $canalordem \subseteq \mathbb{N}_1 \pfun (1 \upto NP \fun \mathbb{N})$

inv9 : $controlecanal \in \mathbb{N}_1 \fun 0 \upto (NP - 1)$

inv10 : $card(GRUPO) \leq NP$

Figura 3.36 – Invariantes do *multicast* causal

A variável $vetorclocks$ representa o relógio vetorial (vetores de *clock*) de cada processo em particular. Para tanto, precisou ser definida como uma função total de 1 até NP, mapeado para outra função total de 1 até NP, mapeado para valores naturais (inv7 da Figura 3.36). A variável $canalordem$, representa a ordem causal dos processos emissores, sendo definida, pela inv8 da Figura 3.36, como um conjunto que deve conter uma função parcial de valores naturais, sem o zero, mapeado para uma função total de 1 até NP, mapeado para valores naturais. Por último, $controlecanal$, representando o número de processos que efetuaram a recepção de determinada mensagem, sendo definida como uma função total de um valor natural, sem o zero, mapeado para valores de zero até NP menos 1 (inv9 da Figura 3.36). A inv10 da Figura 3.36 garante que o grupo *multicast* terá no máximo NP elementos no conjunto.

Fazem parte do refinamento padrão, do padrão *multicast* causal, os seguintes eventos: evento de inicialização, evento $sender$, evento $receiver$ e evento $removemsgcanal$.

O evento de inicialização do modelo, encontra-se na Figura 3.37. Através da Figura 3.37 percebe-se que, em relação variáveis adicionadas neste passo de refinamento: todos os históricos e o $canalordem$ iniciam vazios (act7, act9, act10); a variável usada para associar os históricos é iniciada com valor zero (act8); todos os valores do $vetorclocks$ e da variável $controlecanal$ são mapeado para zero (act11 e act13).

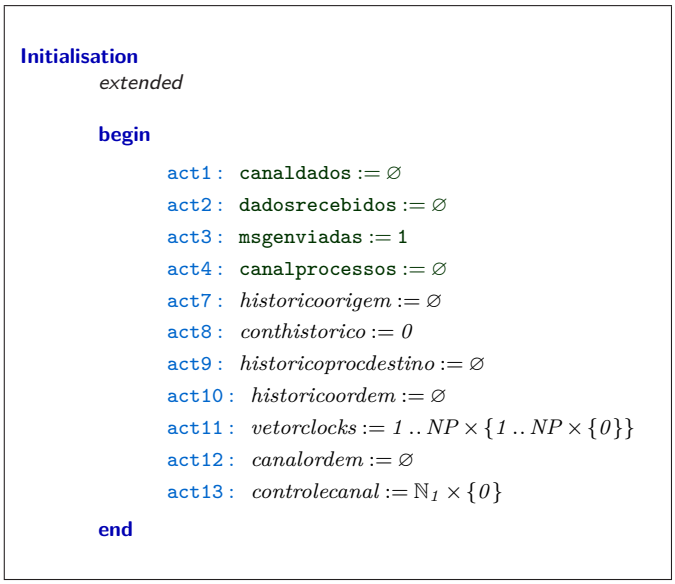


Figura 3.37 – Evento de inicialização do *multicast* causal

Para o evento $sender$ ocorrer, as suas guardas deverão ser respeitadas, ou seja: a mensagem deve pertencer ao conjunto $DADOS$ (grd1 da Figura 3.38); o identificador de associação dos canais deve ser um valor natural (grd2 da Figura 3.38); o processo emissor deve ser um processo válido (grd3 da Figura 3.38); o processo emissor deve possuir um relógio vetorial (grd4 da Figura 3.38); o relógio vetorial do processo emissor, deve conter a ordem causal da mensagem (grd5 da Figura 3.38); a ordem causal deve pertencer aos naturais (grd7 da Figura 3.38); o relógio vetorial do processo emissor deve pertencer a imagem da variável $vetorclocks$ (grd8 da Figura 3.38);

Sendo verdadeiras as guardas, as seguintes ações são aplicadas no modelo: é colocado no canal de comunicação a mensagem a ser enviada para os processos do grupo *multicast* (act1 da Figura 3.38); o identificador responsável por associar os canais é incrementado em uma unidade (act2 da Figura 3.38); na mensagem enviada é associado dados de origem e destinos (act3 da Figura 3.38); atualiza-se a ordem do processo emissor no relógio vetorial (act4 da Figura 3.38); o relógio vetorial do processo emissor é associado a mensagem (act5 da Figura 3.38);

Para este padrão, assim como nos outros padrões *multicast* com ordem (FIFO e total), são dois os eventos responsáveis por realizar a recepção das mensagens: $receiver$ e $removemsgcanal$.

Para o evento $receiver$ ocorrer, as seguintes condições devem ser verdadeiras: deve ter que existir uma mensagem no canal de comunicação (grd1 da Figura 3.39); deve ter que existir dados de origem e destino associado a mensagem (grd2 da Figura 3.39); o processo receptor tem que estar no grupo de destino da mensagem (grd3 da Figura 3.39); o processo emissor tem que ser válido (grd4 da Figura 3.39); o processo receptor tem que pertencer ao grupo *multicast* (grd5 da Figura 3.39); a ordem do processo emissor, no relógio vetorial associado a mensagem ($vetorclocks$) tem quer ser um valor natural (grd6 da Figura 3.39); deve existir um relógio vetorial, do processo emissor, associada a mensagem (grd7 da Figura 3.39); um dos índices do relógio vetorial deve conter a ordem do processo emissor (grd8 da Figura 3.39); o processo receptor deve possuir um relógio vetorial (grd9 da Figura 3.39); o processo emissor deve possuir um índice, com sua ordem, no relógio vetorial do

Event *sender* $\widehat{=}$

refines *sender*

any

$mensagem$
$processoenvia$
$vetortemp$
$ordemprocesso$

where

grd1 : $mensagem \in DADOS$
grd2 : $msgenviadas \in \mathbb{N}_1$
grd3 : $processoenvia \in 1 \mathinner{..} NP$
grd4 : $processoenvia \mapsto vetortemp \in vetorclocks$
grd5 : $processoenvia \mapsto ordemprocesso \in vetortemp$
grd7 : $ordemprocesso \in \mathbb{N}$
grd8 : $vetortemp \in ran(vetorclocks)$

then

act1 : $canaldados := canaldados \cup \{\{msgenviadas \mapsto mensagem\}\}$
act2 : $msgenviadas := msgenviadas + 1$
act3 : $canalprocessos := canalprocessos \cup \{\{msgenviadas \mapsto \{(processoenvia \mapsto GRUPO)\}\}\}$
act4 : $vetorclocks := vetorclocks \lhd \{processoenvia \mapsto (vetortemp \lhd \{processoenvia \mapsto ordemprocesso + 1\})\}$
act5 : $canalordem := canalordem \cup \{\{msgenviadas \mapsto (vetortemp \lhd \{processoenvia \mapsto ordemprocesso + 1\})\}\}$

end

Figura 3.38 – Evento $sender$ do *multicast* causal

processo receptor (grd10 da Figura 3.39); a ordem existente no relógio vetorial da mensagem deve ser uma unidade maior que a existente no relógio vetorial do receptor (grd11 da Figura 3.39); a ordem causal da mensagem deve ser respeitada (grd12 da Figura 3.39); o identificador que associa a mensagem aos outros dados deve ser um valor natural diferente de zero (grd13 da Figura 3.39); a recepção somente é realizada até o penúltimo processo de destino da mensagem em questão (grd14 da Figura 3.39); o relógio logico do processo receptor deve pertencer a imagem de *vetorlclocks* (grd15 da Figura 3.39); o processo receptor tem que ser um processo válido (grd16 da Figura 3.39).

grd1 $\{identmsg \mapsto mensagem\} \in canaldados$
grd2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$
grd3 $processorecv \in grupotemp$
grd4 $processo \in 1 \mathinner{..} NP$
grd5 $processorecv \in GRUPO$
grd6 $ordemprocesso \in \mathbb{N}$
grd7 $\{identmsg \mapsto vetortemp\} \in canalordem$
grd8 $processo \mapsto ordemprocesso \in vetortemp$
grd9 $processorecv \mapsto vetorclocktemp \in vetorclocks$
grd10 $processo \mapsto ordemtemp \in vetorclocktemp$
grd11 $ordemtemp + 1 = ordemprocesso$
grd12 $\forall i, j, k \cdot ((i \mapsto j \in (vetortemp \setminus \{processo \mapsto ordemprocesso\})) \wedge (i \mapsto k \in (vetorclocktemp \setminus \{processo \mapsto ordemtemp\})) \Rightarrow k \geq j)$
grd13 $identmsg \in \mathbb{N}_1$
grd14 $controlecanal(identmsg) < (card(GRUPO) - 2) \wedge controlecanal(identmsg) < NP$
grd15 $vetorclocktemp \in ran(vetorclocks)$
grd16 $processorecv \in 1 \mathinner{..} NP$

Figura 3.39 – Guarda do evento $receiver$ do *multicast* causal

Sendo verdadeiras estas condições, as seguintes ações são aplicadas ao modelo: a *buffer* recebe a mensagem do canal de comunicação (act1 da figura 3.40); o canal da comunicação continua com a mensagem para que os outros processos de destino da mensagem possam receber (act2 da figura 3.40); os históricos são atualizados (act3, act5 e act6 da figura 3.40); o identificador que associa os históricos é incrementado em 1 unidade (act4 da figura 3.40); os dados de origem e destino da mensagem são mantidos no canal (act7 da figura 3.40);a ordem do processo emissor, contida na mensagem, é inserida no relógio vetorial do processo receptor (act8 da figura 3.40); é atualizado o número de processo que receberam a mensagem (act9 da figura 3.40)

act1 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act2 $canaldados := canaldados$
act3 $historicoorigem := historicoorigem \cup \{\{conthistorico \mapsto processo\}\}$
act4 $conthistorico := conthistorico + 1$
act5 $historicoprocdestino := historicoprocdestino \cup \{\{conthistorico \mapsto processorecv\}\}$
act6 $historicoordem := historicoordem \cup \{\{conthistorico \mapsto ordemprocesso\}\}$
act7 $canalprocessos := canalprocessos$
act8 $vetorclocks := vetorclocks \lhd \{processorecv \mapsto (vetorclocktemp \lhd \{processo \mapsto ordemtemp + 1\})\}$
act9 $controlecanal(identmsg) := controlecanal(identmsg) + 1$

Figura 3.40 – ação do evento $receiver$ do *multicast* causal

Analisando a Figura 3.41, que contém as guardas do evento $removemsgcanal$, nota-se que para o evento ocorrer são necessárias as mesmas restrições impostas pelo evento $receiver$, exceto pela gr14 da Figura 3.41, que no evento $removemsgcanal$, determina que a recepção somente ocorrerá quanto existir apenas um processo faltando para receber a mensagem.

grd1 $\{identmsg \mapsto mensagem\} \in canaldados$
grd2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$
grd3 $processorecv \in grupotemp$
grd4 $processo \in 1..NP$
grd5 $processorecv \in GRUPO$
grd6 $ordemprocesso \in \mathbb{N}$
grd7 $\{identmsg \mapsto vetortemp\} \in canalordem$
grd8 $processo \mapsto ordemprocesso \in vetortemp$
grd9 $processorecv \mapsto vetorclocktemp \in vetorclocks$
grd10 $processo \mapsto ordemtemp \in vetorclocktemp$
grd11 $ordemtemp + 1 = ordemprocesso$
grd12 $\forall i, j, k \cdot ((i \mapsto j \in (vetortemp \setminus \{processo \mapsto ordemprocesso\})) \wedge (i \mapsto k \in (vetorclocktemp \setminus \{processo \mapsto ordemtemp\})) \Rightarrow k \geq j)$
grd13 $identmsg \in \mathbb{N}_1$
grd14 $identmsg \in \mathbb{N}_1 \wedge controlecanal(identmsg) = (card(GRUPO) - 2)$
grd15 $vetorclocktemp \in ran(vetorclocks)$
grd16 $processorecv \in 1..NP$

Figura 3.41 – guarda do evento $removemsgcanal$ do *multicast* causal

Portanto sendo verdadeiras as guardas do evento $removemsgcanal$, as seguintes ações são realizadas: a *buffer* recebe a mensagem do canal de comunicação (act1 da Figura 3.42); a mensagem é removida do canal de comunicação (act2 da Figura 3.42); os históricos são atualizados com os dados associados a mensagem (act3, act5, act6 da Figura 3.42); o identificador que associa os históricos é incrementado em uma unidade (act4 da Figura 3.42); os dados de origem e destino são removidos do canal (act7 da Figura 3.42); o relógio vetorial do processo receptor é atualizado para a ordem do processo originador, contida no relógio vetorial associado a mensagem (act8 da Figura

3.42); indica-se a realização da ultima recepção para a mensagem em questão (act9 da Figura 3.42); o relógio vetorial associado a mensagem é removido do canal (act12 da Figura 3.42).

act1 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act2 $canaldados := canaldados \setminus \{\{identmsg \mapsto mensagem\}\}$
act3 $historicoorigem := historicoorigem \cup \{\{conthistorico \mapsto processo\}\}$
act4 $conthistorico := conthistorico + 1$
act5 $historicoprocdestino := historicoprocdestino \cup \{\{conthistorico \mapsto processorecv\}\}$
act6 $historicoordem := historicoordem \cup \{\{conthistorico \mapsto ordemprocesso\}\}$
act7 $canalprocessos := canalprocessos \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}$
act8 $vetorclocks := vetorclocks \lhd\!\!+ \{processorecv \mapsto (vetorclocktemp \lhd\!\!+ \{processo \mapsto ordemtemp + 1\})\}$
act9 $controlecanal(identmsg) := controlecanal(identmsg) + 1$
act10 $canalordem := canalordem \setminus \{\{identmsg \mapsto vetortemp\}\}$

Figura 3.42 – ação do evento $removemsgcanal$ do *multicast* causal

Dada a modelagem do multicast Causal, nota-se que no refinamento padrão:

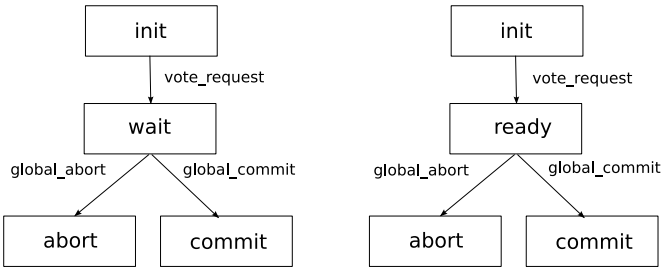
- foi necessário introduzir uma estrutura de controle para cada processo manter seu relógio vetorial ($vetorclocks$).
- foi adicionado um canal de controle ($canalordem$), contendo informação sobre a relação causal entre as mensagens, na forma de um relógio vetorial. Este canal de comunicação é associado a mensagem com um identificador único.
- a garantia da ordenação causal no sistema é comprovada pelas restrição das guardas nos eventos receptores. As guardas dos eventos receptores foram restritas para realizar a recepção das mensagem em que: a ordem do processo originador, contida no relógio vetorial associado a mensagem, for uma unidade maior que ordem do processo originador, presente no relógio vetorial do processo receptor (grd11 das Figuras 3.39 e 3.41); o restante dos processos (diferentes do originador), presentes no relógio vetorial da mensagem, devem possuir uma ordem maior ou igual as contidas no relógio vetorial do processo receptor (grd12 das Figuras 3.39 e 3.41).
- foi adicionado um evento ($removemsgcanal$) para remoção da mensagem do canal de comunicação e os dados associados a mensagem. A remoção ocorre na recepção do último processo, de um dada mensagem, por um grupo *multicast*. Desta forma, removendo-se o não determinísmo do sistema.

4. ESTUDO DE CASO

Para exemplificação do uso da biblioteca de padrões (Capítulo 3), neste capítulo, foi realizada a modelagem de um sistema distribuído em Event-B: o protocolo *two-phase commit* (2PC).

O 2PC é um protocolo para efetivação de modificações por um grupo de processos, que opera em duas fases. Em ambas fases, um processo, dito líder, coordena a ações dos demais processos, ditos participantes.

No 2PC, o líder e os processos participantes iniciam no estado *init* (Figura 4.1).



(a) Fases do líder

(b) Fases dos processos participantes

Figura 4.1 – *two phase commit*

Na primeira fase são realizadas as seguintes ações:

1. o líder muda seu estado para *wait*, enviando, para os processos participantes, uma mensagem de *vote_request* (Figura 4.1(a)). Desta forma convocando os processos participantes a votarem por efetivar (*commit*) ou abortar (*abort*) modificações conjuntas.

2. os processos participantes votam, mudando os seus estados para *ready* (Figura 4.1(b)).

3. o líder recolhe os votos dos participantes e computa a decisão :

   (a) caso todos tenham votado pela efetivação, o líder modifica seu estado para *commit* (Figura 4.1(a)).

   (b) caso ao menos um participante tenha votado por abortar as modificações, o líder modifica seu estado para *abort* (Figura 4.1(a)).

Na segunda fase, o líder informa a decisão a todos participantes: caso seja pela efetivação, o líder envia uma mensagem de *global_commit* para os processos participantes; caso seja por abortar, o líder envia uma mensagem de *global_abort*.

Por fim os processos participantes confirmam a decisão, mudando seus estados com base na decisão do líder, ou seja, *commit* ou *abort* (Figura 4.1(b)).

A modelagem do 2PC contém um contexto que é parametrizado para todas as máquinas do modelo. No contexto foram definidos quatro conjuntos: *FASELIDER*, *FASEPROCESSO*, *MSGLIDER* e *VOTOS*.

FASELIDER contém, como elementos do conjunto, os estados válidos para o processo líder (Figura 4.1(a)). Para tanto, no axioma axm1, da Figura 4.2, foram definidas as seguintes constantes como elementos do conjunto: *init*, *wait*, *abort*, *commit*. Já *FASEPROCESSO* contém, como elementos do conjunto, os estados válidos para os processos participantes (Figura 4.1(b)), portanto possuíndo as seguintes constantes como elementos do conjunto (axm2 da Figura 4.2): *initp*, *ready*, *abortp*, *commitp*.

axm1 $FASELIDER = \{init, wait, abort, commit\}$
axm2 $FASEPROCESSO = \{initp, ready, abortp, commitp\}$
axm3 $init \neq wait \wedge init \neq abort \wedge init \neq commit \wedge wait \neq abort \wedge wait \neq commit \wedge abort \neq commit$
axm4 $initp \neq ready \wedge initp \neq abortp \wedge initp \neq commitp \wedge ready \neq abortp \wedge ready \neq commitp \wedge abortp \neq commitp$
axm5 $NP = 4$
axm6 $PROCESSOS = \{2, 3, 4\}$
axm7 $LIDER = 1$
axm8 $MSGLIDER = \{vote\_request, vote\_commit, vote\_abort\}$
axm9 $vote\_request \neq vote\_commit \wedge vote\_request \neq vote\_abort \wedge vote\_commit \neq vote\_abort$
axm10 $VOTOS = \{semvoto, processocommit, processoabort\}$
axm11 $semvoto \neq processoabort \wedge semvoto \neq processocommit \wedge processoabort \neq processocommit$

Figura 4.2 – contexto do 2PC

MSGLIDER contém, como elementos do conjunto, as seguintes constantes (axm8 da Figura 4.2): *vote_request*, *vote_commit*, *vote_abort*. *MSGLIDER* representa o conjunto de mensagens que o líder pode enviar para os processos participantes. Já o conjunto *VOTOS*, que representa os possíveis votos dos processos participantes, é definido pelo axm10, da Figura 4.2, com as seguintes constantes como elementos: *semvoto*, *processoscommit*, *processoabort*.

Além das constantes, citadas anteriomente, como elementos dos conjuntos, o contexto do modelo contém também as seguintes constantes: *LIDER* que representa o processo líder na modelagem, sendo definida com valor 1 (axm7 da Figura 4.2); *NP* que representa o valor máximo de processos na modelagem, sendo definido com o valor 4; *PROCESSOS* representando o conjunto dos processos participantes, sendo defindo no axioma axm6, da Figura 4.2, com os valores 2, 3 e 4 como elementos de *PROCESSOS*.

Os axiomas axm3, axm4, axm9, axm11 são utilizados para especificar na modelagem que os diversos elementos contidos nos conjuntos, anteriomente descritos, são diferentes.

A modelagem do 2PC apresenta cinco máquinas, chamados a seguir de 2PC_1..5, e quatro refinamentos. Para tal modelagem, a comunicação necessária entre os processos participantes e o líder, foi incorporada através da aplicação de dois padrões da biblioteca de padrões: padrão *multicast* FIFO (Seção 3.6) e padrão *unicast* síncrono (Seção 3.2).

4.1 2PC_1

A modelagem do 2PC_1, inicia contemplando a comunicação no sentido líder - participantes e estruturas que definem os estados de cada processo. Com relação aos padrões anteriormente

descritos, esta máquina tem combinação com a especificação padrão do *multicast* sem ordem (Seção 3.5). Assim, esta máquina, contempla canal de comunicação e canal de controle com identificação de originador e destinatário, além de um evento de envio e um de recepção.

As seguintes variáveis compõem a máquina 2PC_1: *estadoprocesso*, *estadolider*, *canallider*, *canalcontrole*, *dadosrecebidos*, *msgenviadas*.

INVARIANTS

inv1 : $estadoprocesso \in 2 \mathinner{..} NP \rightarrow FASEPROCESSO$
inv2 : $estadolider \in FASELIDER$
inv3 : $canallider \subseteq \mathbb{N}_1 \pfun MSGLIDER$
inv4 : $canalcontrole \subseteq (\mathbb{N}_1 \pfun ((1 \mathinner{..} NP) \pfun \mathbb{P}(1 \mathinner{..} NP)))$
inv5 : $dadosrecebidos \subseteq \mathbb{N}_1 \pfun MSGLIDER$
inv6 : $msgenviadas \in \mathbb{N}_1$

Figura 4.3 – Invariantes do 2PC_1

A variável *estadoprocesso* representa o estado que cada processo participante em particular encontra-se. Tal variável foi defindа como uma função total de 2 até *NP*, mapeada para o conjunto *FASEPROCESSO* (inv1 da Figura 4.3). A variável *ESTADOLIDER* representa o estado que o líder encontra-se, portanto sendo definida como um elemento de *FASELIDER* (inv2 da Figura 4.3).

As variáveis *canallider*, *canalcontrole*, *dadosrecebidos* e *msgenviadas* são variáveis responsáveis pelo envio das mensagens do líder para os processos participantes, desta forma, necessárias para a combinação do modelo *two phase commit* com o padrão *multicast* FIFO. Portanto as invariantes inv3, inv4, inv5 e inv6 são definidas de acordo com a especificação padrão, do padrão (Seção 3.5).

No 2PC_1, todos os processo participantes iniciam no estado *initp* e o líder inicia no estado *init* (act1 e act2 da Figura 4.4). O restante das ações (act3, act4, act5, act6 da Figura 4.4) são necessárias para combinação, portanto sendo inicializadas de acordo com a especificação padrão, do padrão *multicast* FIFO (Seção 3.5).

Initialisation

begin

act1 : $estadoprocesso := 2 \mathinner{..} NP \times \{initp\}$
act2 : $estadolider := init$
act3 : $canallider := \varnothing$
act4 : $canalcontrole := \varnothing$
act5 : $dadosrecebidos := \varnothing$
act6 : $msgenviadas := 1$

end

Figura 4.4 – Evento de inicialização da máquina 2PC_1

Além do evento de inicialização (Figura 4.4), o 2PC_1 é composto de dois eventos: o evento

de envio, das mensagens *multicast*, do líder para os participantes, chamado *multicastlider*; o evento de recepção, das mensagens *multicast* pelos participantes, chamado *processorecebemsg*.

No evento *multicastlider* (Figura 4.5) foram adicionadas: as guardas necessárias para a combinação do 2PC_1 com o padrão (grd1, grd2 e grd4); uma restrição nas guardas, afim de garantir que somente o líder envie mensagens *multicast* para os processos participantes (grd3); uma guarda para restringir as trocas de estado do líder, para um estado válido do modelo (grd5); ações responsáveis por colocar a mensagem no canal, sendo necessárias para a combinação do padrão *multicast* FIFO (act2, act3 e act4); uma ação responsável por modificar o estado do líder (act1).

Event $multicastlider \mathrel{\widehat{=}}$

any

$mensagem$
$processoenvia$
$estadotemp$

where

grd1 : $mensagem \in MSGLIDER$
grd2 : $processoenvia \in 1 \mathinner{..} NP$
grd3 : $processoenvia = LIDER$
grd4 : $msgenviadas \in \mathbb{N}_1$
grd5 : $estadotemp \in FASELIDER$

then

act1 : $estadolider := estadotemp$
act2 : $canallider := canallider \cup \{\{msgenviadas \mapsto mensagem\}\}$
act3 : $msgenviadas := msgenviadas + 1$
act4 : $canalcontrole := canalcontrole \cup \{\{msgenviadas \mapsto \{(processoenvia \mapsto PROCESSOS)\}\}\}$

end

Figura 4.5 – Evento *multicastlider* 2PC_1

Já no evento *processorecebemsg* foram adicionadas: as guardas necessárias para a combinação (grd1, grd2 e grd3 da Figura 4.6); as guardas responsáveis por restringir a troca dos processos para um estado válido do modelo (gdr4 e grd5 da Figura 4.6); as ações responsáveis por receber as mensagens do canal de comunicação, sendo necessárias para combinação do modelo com o padrão *multicast* FIFO (act2, act3 act4 da Figura 4.6); a ação responsável pela troca de estado dos processos participantes na recepção das mensagens (act1 da Figura 4.6).

Dada a modelagem, percebe-se que o envio das mensagens do líder para os processos participantes e as mundanças de estado da modelagem, tanto do líder como dos participantes, ocorrem sem ordem. Além disso, como comentado anteriomente, as comunicação até a presente modelagem ocorrem em apenas um sentido (líder - processos participantes).

Para aplicar o padrão na modelagem, e portanto, ordenar a comunicação do líder para os processos participantes, foram necessárias as três fase do padrão: combinação, verificação e incorporação.

Event $processosrecebemmsg \mathrel{\widehat{=}}$

any

$identmsg$
$mensagem$
$processo$
$grupotemp$
$processorecv$
$estadotemp$

where

grd1 : $\{identmsg \mapsto mensagem\} \in canallider$
grd2 : $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalcontrole$
grd3 : $processorecv \in grupotemp$
grd4 : $processorecv \in dom(estadoprocesso)$
grd5 : $estadotemp \in FASEPROCESSO$

then

act1 : $estadoprocesso(processorecv) := estadotemp$
act2 : $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act3 : $canalcontrole :\in \{canalcontrole, canalcontrole \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}\}$
act4 : $canallider :\in \{canallider, (canallider \setminus \{\{identmsg \mapsto mensagem\}\})\}$

end

Figura 4.6 – Evento *processorecebemsg* 2PC_1

4.1.1 Combinação

Nesta fase, foram combinadas as variáveis e os eventos da máquina. Para tanto, foram utilizados o 2PC_1, anteriomente descrito, e a especificação padrão do *multicast FIFO* (Seção 3.5).

Na Tabela 4.1 encontra-se as variáveis combinadas e as variáveis extras da máquina.

Tabela 4.1 – Passo C1

	Variáveis Combinadas ($\rightsquigarrow$)	
Linha	Especificação padrão	2PC_1
1	$canaldados$	$canalider$
2	$dadosrecebidos$	$dadosrecebidos$
3	$msgenviadas$	$msgenviadas$
4	$canalprocessos$	$canalcontrole$
	Variáveis não combinadas (extras)	
5	$estadoprocesso$	
6	$estadolider$	

Observando a Tabela 4.1 percebe-se que: na linha 1, *canaldados* foi combinada com *canallider*; na linha 2, *dadosrecebidos* (especificação padrão) foi combinada com *dadosrecebidos* (2PC_1); na linha 3, *msgenviadas* (especificação padrão) foi combinada com *msgenviadas* (2PC_1); na linha 4, *canalprocessos* foi combinada com *canalcontrole*; *estadoprocesso* e *estadolider* são variáves extras na combinação (linhas 5 e 6).

Na Tabela 4.2 são apresentados os eventos combinado. Observando a Tabela 4.2, percebe-se:

na linha 1, foram combinados o *initialisation* da especificação padrão e o *initialisation* do 2PC_1; na linha 2, foram combinados o *sender* da especificação padrão e o *multicastlider* do 2PC_1; na linha 3, foram combinados o *receiver* da especificação padrão e o *processorecebemsg* do 2PC_1; nesta combinação, não existem eventos extras.

Tabela 4.2 – Passo C2

	Eventos Combinados ($\rightsquigarrow$)	
Linha	Especificação padrão	2PC
1	$initialisation$	$initialisation$
2	$sender$	$multicastlider$
3	$receiver$	$processorecebemsg$

Sendo definidas as variáveis combinadas e os eventos combinados é terminada a primeira fase (combinação). Na segunda fase é realizada a verificação das variáveis e eventos combinados.

4.1.2 Verificação

Para realização da fase de verificação, foram utilizados o 2PC_1, anteriomente descrito, e a especificação padrão do *multicast FIFO* (Seção 3.5).

Na Figura 4.7 é descrita a comparação sintática das guardas dos eventos combinados da especificação padrão (lado esquerdo da Figura 4.7), com as guardas do evento do 2PC_1 (lado direito da Figura 4.7).

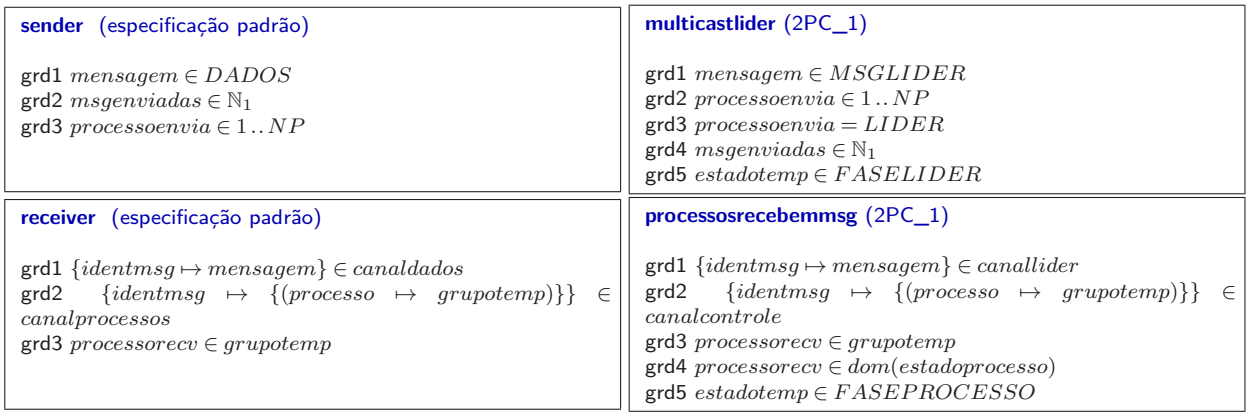


Figura 4.7 – passo V1

Analisando a Figura 4.7, nota-se que: as guardas grd1, grd2, grd3 do *sender* são sintaticamente iguais, respectivamente, às guardas grd1, grd4, grd2 do evento *multicastlider*; as guardas grd1, grd2, grd3 do *receiver* são sintaticamente iguais, respectivamente, às guardas grd1, grd2, grd3 do *processosrecebemmsg*.

No passo V1, o evento de inicialização não é comparado por se tratar de um evento sem guardas.

A verificação V1 é dada como correta, visto que todos os eventos combinados da especificação padrão, do padrão *multicast* FIFO, possuem guardas sintaticamente iguais nos eventos combinados do 2PC_1.

No passo V2 é verificado a existencia de guardas extras. Observando a Figura 4.7, percebe-se a existência de quatro guardas extras (não combinadas): grd3 e grd5 do evento *multicastlider*; grd4 e grd5 do evento *processosrecebemsg*.

Definidas as guardas extras da máquina, foi realizado o passo V3, ou seja, a verificação sintática das ações (Figura 4.8).

initialisation (especificação padrão)

act1 $canaldados := \varnothing$
act2 $dadosrecebidos := \varnothing$
act3 $msgenviadas := 1$
act4 $canalprocessos := \varnothing$

initialisation (2PC_1)

act1 $estadoprocesso := 2..NP \times \{initp\}$
act2 $estadolider := init$
act3 $canallider := \varnothing$
act4 $canalcontrole := \varnothing$
act5 $dadosrecebidos := \varnothing$
act6 $msgenviadas := 1$

sender (especificação padrão)

act1 $canaldados := canaldados \cup \{\{msgenviadas \mapsto mensagem\}\}$
act2 $msgenviadas := msgenviadas + 1$
act3 $canalprocessos := canalprocessos \cup \{\{msgenviadas \mapsto \{(processoenvia \mapsto GRUPO)\}\}\}$

multicastlider (2PC_1)

act1 $estadolider := estadotemp$
act2 $canallider := canallider \cup \{\{msgenviadas \mapsto mensagem\}\}$
act3 $msgenviadas := msgenviadas + 1$
act4 $canalcontrole := canalcontrole \cup \{\{msgenviadas \mapsto \{(processoenvia \mapsto PROCESSOS)\}\}\}$

receiver (especificação padrão)

act1 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act2 $canaldados :\in \{canaldados, canaldados \setminus \{\{identmsg \mapsto mensagem\}\}\}$
act3 $canalprocessos :\in \{canalprocessos, canalprocessos \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}\}$

processosrecebemmsg (2PC_1)

act1 $estadoprocesso(processorecv) := estadotemp$
act2 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$
act3 $canalcontrole :\in \{canalcontrole, canalcontrole \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}\}$
act4 $canallider :\in \{canallider, (canallider \setminus \{\{identmsg \mapsto mensagem\}\})\}$

Figura 4.8 – passo V3

Verificando os eventos da especificação padrão (lado esquerdo da Figura 4.8) e do 2PC_1 (lado direito da Figura 4.8), percebe-se que: as ações act1, act2, act3 e act4, do evento *initialisation* da especificação padrão, são sintaticamente iguais, respectivamente, às ações act3, act5, act6 e act4 do evento *initialisation* do 2PC_1 (Figura 4.8); no evento *sender* a ações act1, act2, act3 são sintaticamente iguais, respectivamente, às ações act2, act3, act4 do evento *multicastlider* (Figura 4.8);no evento *receiver* a ações act1, act2, act3 são sintaticamente iguais, respectivamente, às ações act2, act4, act3 do evento *processosrecebemsg* (Figura 4.8).

Todas as ações dos eventos *initialisation*, *sender* e *receiver* (especificação padrão), tem uma ação sintaticamente igual nos respectivos eventos combinados do 2PC_1 (Tabela 4.2). Portanto é dada como correta a verificação V3.

No último passo da verificação (passo V4), procura-se ações extras (não combinadas) nos eventos combinados do 2PC_1. Neste passo, analisando a Figura 4.8, foram encontradas quatro ações extras no 2PC_1: act1 e act2 do evento *initialisation*; act1 do evento *multicastlider*; act1 do evento *processosrecebemsg*.

Completado os passos de verificação, percebe-se que o padrão *multicast* pode ser incorporado no 2PC_1. Para incorporação do padrão é gerado um refinamento no modelo *two phase commit*, chamado 2PC_2.

4.2 2PC_2

No 2PC_2 é incorporado ao modelo a ordenação FIFO. A aplicação do pardão FIFO gera um modelo análogo ao anterior, unicamente inserindo a propriedade de ordem na comunicação lider - participante.

A ordenação FIFO é necessária na comunicação líder - participante, pois uma decisão de abortar poderia ser recebida antes de um participante ter recebido a mensagem de pedido de votação. Isto poderia acontecer caso um outro participante imediatamente vote por abortar.

Para incorporação é utilizado o refinamento padrão, do padrão multicast FIFO (Seção 3.6), e o 2PC_1 (Seção 4.1).

No primeiro passo da incorporação, foram copiadas as variáveis do refinamento padrão (Seção 3.6) e as variáveis extras (Tabela 4.1) para a máquina 2PC_2 (passo I1.1 da Tabela 4.3).

Segundo [Für09] é considerado um bom método renomear as variáveis criadas do refinamento padrão, para que numa futura aplicação, do mesmo padrão no modelo, não existam variáveis repetidas. Para tanto, após copiar as variáveis para a máquina, foram inseridas, nas variáveis criadas no refinamento padrão, o sufixo *_votacao* (passo I1.2 da Tabela 4.3).

Além disso, como relatado anteriomente (Seção 2.2), para que os teoremas identifiquem as variáveis, foi necessário modificar as variáveis combinadas da especificação padrão, existentes no refinamento padrão, pelas variáveis combinadas do 2PC_1 (passo I1.3 da Tabela 4.3). Desta forma ficando as variáveis da máquina 2PC de acordo com o passo I1.3 da Tabela 4.3.

Tabela 4.3 – Passo I1

Váriaveis do 2PC_2		
Passo I.1		
estadoprocesso	*estadoprocesso*	*estadoprocesso*
estadolider	*estadolider*	*estadolider*
canaldados	*canaldados*	**canalider**
dadosrecebidos	*dadosrecebidos*	**dadosrecebidos**
msgenviadas	*msgenviadas*	**msgenviadas**
canalprocessos	*canaprocessos*	**canalcontrole**
canalcontrole	**canalcontrole_votacao**	*canalcontrole_votacao*
historicosend	**historicosend_votacao**	*historicosend_votacao*
historicoorigem	**historicoorigem_votacao**	*historicoorigem_votacao*
conthistorico	**conthistorico_votacao**	*conthistorico_votacao*
historicoprocdestino	**historicoprocdestino_votacao**	*historicoprocdestino_votacao*
historicoordem	**historicoordem_votacao**	*historicoordem_votacao*
controleordem	**controleordem_votacao**	*controleordem_votacao*
controlecanal	**controlecanal_votacao**	*controlecanal_votacao*

Definidas as variáveis da máquina, foi realizada a incorporação das invariantes do 2PC_2 (passo I2). Para tanto, foram copiadas as invariantes da especificação padrão (Seção 3.5) e do refinamento padrão (Seção 3.6), do padrão multicast FIFO, para o refinamento 2PC_2 (Figura 4.9).

Observando a Figura 4.9 é possivel notar que as mudanças necessárias as variáveis, no passo I1, foram refletidas nas invariantes do 2PC_1, assim como no decorrer da modelagem, serão refletidas na construção dos eventos.

Copiadas as invariantes para o 2PC_1, o próximo passo é a incorporação dos eventos da modelagem (passo I3, I4 e I5).

INVARIANTS

inv1 : $canallider \subseteq \mathbb{N}_1 \pfun MSGLIDER$

inv2 : $canalcontrole \subseteq (\mathbb{N}_1 \pfun ((1 \upto NP) \pfun \mathbb{P}(1 \upto NP)))$

inv3 : $dadosrecebidos \subseteq \mathbb{N}_1 \pfun MSGLIDER$

inv4 : $msgenviadas \in \mathbb{N}_1$

inv5 : $canalcontrole\_votacao \subseteq \mathbb{N}_1 \pfun \mathbb{N}$

inv6 : $historicosend\_votacao \in 1 \upto NP \to \mathbb{N}$

inv7 : $historicoprocdestino\_votacao \subseteq \mathbb{N} \pfun PROCESSOS$

inv8 : $historicoordem\_votacao \subseteq \mathbb{N} \pfun \mathbb{N}$

inv9 : $controleordem\_votacao \in 1 \upto NP \to (1 \upto NP \to \mathbb{N})$

inv10 : $controlecanal\_votacao \in \mathbb{N}_1 \to 0 \upto NP$

inv11 : $historicoorigem\_votacao \subseteq \mathbb{N} \pfun 1 \upto NP$

inv12 : $conthistorico\_votacao \in \mathbb{N}$

inv13 : $card(PROCESSOS) \leq NP$

inv14 : $\forall x \cdot (x \in dom(controlecanal\_votacao) \Rightarrow controlecanal\_votacao(x) \leq card(PROCESSOS))$

Figura 4.9 – invariantes do refinamento 2PC_2 (passo I2)

No refinamento padrão do padrão multicast FIFO (Seção 3.6) existem 4 eventos: *initialisation*, *sender*, *receiver* e o *removemsgcanal*.

O evento *receiver* da especificação padrão do *multicast* FIFO foi decomposto em dois eventos no refinamento padrão: *receiver* e *removemsgcanal*. Com isso, o evento *removemsgcanal*, apesar de surgir no refinamento padrão, não é considerado um evento novo, mas um evento refinado da especificação padrão.

Dado que o refinamento padrão, do *multicast* FIFO, é apenas constituído de eventos refinados dos eventos combinados da especificação padrão e dado que não existam eventos extras na combinação (Seção 4.1.1), os passos I3 e I4 da incorporação não foram necessárias.

No passo I5 é realizada a incorporação dos eventos refinados da especificação padrão, presentes no refinamento padrão, do padrão *multicast* FIFO. Para tal passo, a incorporação é realizada evento a evento, ou seja, os eventos *initialisation*, *multicastlider*, *processosrecebemsg* e *removemsg_votacao*, do 2PC_2, foram incorporados um a um.

O primeiro evento incorporado no 2PC_2 foi o *initialisation*. Dado que este evento não contém guardas os passos I5.1 e I5.2 não foram aplicados. Portanto, para este evento, necessitando apenas a incorporação das ações do evento *initialisation*.

Primeiramente compiaram-se as ações do evento *initialisation* do refinamento padrão, para a máquina 2PC_2 (passo I5.3 da Figura 4.10). Em seguida, foram acrescentadas a ações extras ao evento (linha 1 e 2 do passo I5.4 da Figura 4.10) e realizadas as trocas necessárias nas ações, refletindo as trocas de variáveis do passo I1. Portanto ficando as ações do evento de inicialização do 2PC_2 apresentadas no passo I5.4 da Figura 4.10.

O segundo evento incorporado no 2PC_2 foi o evento *multicastlider*. Neste evento, ao contrário do evento de inicialização, foram incorporadas ações e guardas.

Passo I5.3 do *initialisation*	Passo I5.4 do *initialisation*
1 $canaldados := \varnothing$ 2 $dadosrecebidos := \varnothing$ 3 $msgenviadas := 1$ 4 $canalprocessos := \varnothing$ 5 $canalcontrole := \varnothing$ 6 $historicosend := 1..NP \times \{0\}$ 7 $historicoorigem := \varnothing$ 8 $conthistorico := 0$ 9 $historicoprocdestino := \varnothing$ 10 $historicoordem := \varnothing$ 11 $controleordem := 1..NP \times \{1..NP \times \{0\}\}$ 12 $controlecanal := \mathbb{N}_1 \times \{0\}$	1 $estadoprocesso := 2..NP \times \{initp\}$ 2 $estadolider := init$ 3 $canallider := \varnothing$ 4 $canalcontrole := \varnothing$ 5 $dadosrecebidos := \varnothing$ 6 $msgenviadas := 1$ 7 $canalcontrole\_votacao := \varnothing$ 8 $historicosend\_votacao := 1..NP \times \{0\}$ 9 $historicoorigem\_votacao := \varnothing$ 10 $conthistorico\_votacao := 0$ 11 $historicoprocdestino\_votacao := \varnothing$ 12 $historicoordem\_votacao := \varnothing$ 13 $controleordem\_votacao := 1..NP \times \{1..NP \times \{0\}\}$ 14 $controlecanal\_votacao := \mathbb{N}_1 \times \{0\}$

Figura 4.10 – passo I5 do evento *initialisation*

Na incorporação das guardas foram copiadas as guardas do evento *sender* do refinamento padrão, do padrão *multicast* FIFO (passo I5.1 da Figura 4.11). Após tal passo, foram adiconadas as guardas extras (linhas 3 e 5 do passo I5.2 da Figura 4.11) e realizada a modificação necessária na guarda da linha 1 do passo I5.1 da figura 4.11.

Passo I5.1 do *multicastlider*	Passo I5.2 do *multicastlider*
1 $mensagem \in DADOS$ 2 $msgenviadas \in \mathbb{N}_1$ 3 $processoenvia \in 1..NP$	1 $mensagem \in MSGLIDER$ 2 $processoenvia \in 1..NP$ 3 $processoenvia = LIDER$ 4 $msgenviadas \in \mathbb{N}_1$ 5 $estadotemp \in FASELIDER$

Figura 4.11 – passo I5.1 e I5.2 do evento *multicastlider*

Na incorporação das ações do evento *multicastlider*: copiaram-se as ações do evento *sender* para a máquina 2PC_2 (passo I5.3 da Figura 4.12); foi adicionado uma ação extra no evento (linha 1 do passo 5.4 da Figura 4.12); realizaram-se as trocas necessárias nas ações do evento (linhas 1, 3, 4 e 5 do passso I5.3 da Figura 4.12), para refletir as mudanças das variáveis, ocorridas no passo I1.

Passo I5.3 do *multicastlider*	Passo I5.4 do *multicastlider*
1 $canaldados := canaldados \cup \{\{msgenviadas \mapsto mensagem\}\}$ 2 $msgenviadas := msgenviadas + 1$ 3 $canalprocessos := canalprocessos \cup \{\{msgenviadas \mapsto \{(processoenvia \mapsto GRUPO)\}\}\}$ 4 $canalcontrole := canalcontrole \cup \{\{msgenviadas \mapsto (historicosend(processoenvia) + 1)\}\}$ 5 $historicosend(processoenvia) := historicosend(processoenvia) + 1$	1 $estadolider := estadotemp$ 2 $canallider := canallider \cup \{\{msgenviadas \mapsto mensagem\}\}$ 3 $msgenviadas := msgenviadas + 1$ 4 $canalcontrole := canalcontrole \cup \{\{msgenviadas \mapsto \{(processoenvia \mapsto PROCESSOS)\}\}\}$ 5 $canalcontrole\_votacao := canalcontrole\_votacao \cup \{\{msgenviadas \mapsto (historicosend\_votacao(processoenvia) + 1)\}\}$ 6 $historicosend\_votacao(processoenvia) := historicosend\_votacao(processoenvia) + 1$

Figura 4.12 – passo I5.3 e I5.4 do evento *multicastlider*

Para incorporação do evento *processosrecebemmsg*, foram copiadas as guardas do evento *receiver* do refinamento padrão (passo I5.1 da Figura 4.13) e a seguir foram adicionadas as guardas extras do evento *processosrecebemmsg* do 2PC_1. Além disso, foram realizadas as mudanças necessárias as guardas. As guardas do evento *processosrecebemsg* podem ser visualizada no passo I5.2 da Figura 4.13.

Passo I5.1 do *processorecebemmsg*	Passo I5.2 do *processorecebemmsg*
1 $\{identmsg \mapsto mensagem\} \in canaldados$ 2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$ 3 $processorecv \in grupotemp$ 4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole$ 5 $processo \in 1..NP$ 6 $processorecv \in GRUPO$ 7 $ordemprocesso \in \mathbb{N}_1$ 8 $processorecv \mapsto conjuntotemp \in controleordem$ 9 $processo \mapsto ordemtemp \in conjuntotemp$ 10 $ordemprocesso = ordemtemp + 1$ 11 $identmsg \in dom(controlecanal)$ 12 $controlecanal(identmsg) < (card(GRUPO) - 1)$ $\wedge controlecanal(identmsg) < NP$ 13 $processorecv \in 1..NP$ 14 $conjuntotemp \in ran(controleordem)$	1 $\{identmsg \mapsto mensagem\} \in canallider$ 2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalcontrole$ 3 $processorecv \in grupotemp$ 4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole\_votacao$ 5 $processo \in 1..NP$ 6 $processorecv \in PROCESSOS$ 7 $ordemprocesso \in \mathbb{N}_1$ 8 $processorecv \mapsto conjuntotemp \in controleordem\_votacao$ 9 $processo \mapsto ordemtemp \in conjuntotemp$ 10 $ordemprocesso = ordemtemp + 1$ 11 $identmsg \in dom(controlecanal\_votacao)$ 12 $controlecanal\_votacao(identmsg) <$ $(card(PROCESSOS) - 1) \wedge$ $controlecanal\_votacao(identmsg) < NP$ 13 $processorecv \in 1..NP$ 14 $conjuntotemp \in ran(controleordem\_votacao)$ 15 $estadotemp \in FASEPROCESSO$

Figura 4.13 – passo I5.1 e I5.2 do evento *processosrecebemmsg*

No passo I5.3 e I5.4 do evento *processosrecebemmsg*, foram incorporadas as ações, ou seja: compiaram-se as ações do evento *receiver*, do refinamento padrão para o evento *processorecebemmsg* (I5.3 da Figura 4.14); foram adicionadas as variáveis extras no evento *processorecebemmsg* (linha 1 do passo I5.4 da Figura 4.14); As mudanças nas variáveis do 2PC_2 foram refletidas no evento (linha 1 até linha 9 do passo I5.3 da Figura 4.14).

Na Figura 4.14 (passo I5.4) é apresentada as ações do evento *processorecebemmsg* do 2PC_2.

Passo I5.3 do *processorecebemmsg*	Passo I5.4 do *processorecebemmsg*
1 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$ 2 $canaldados := canaldados$ 3 $historicoorigem := historicoorigem \cup \{\{conthistorico \mapsto processo\}\}$ 4 $conthistorico := conthistorico + 1$ 5 $historicoprocdestino := historicoprocdestino \cup \{\{conthistorico \mapsto processorecv\}\}$ 6 $historicoordem := historicoordem \cup \{\{conthistorico \mapsto ordemprocesso\}\}$ 7 $controleordem := controleordem \lhd \{processorecv \mapsto (conjuntotemp \lhd \{processo \mapsto ordemprocesso\})\}$ 8 $canalprocessos := canalprocessos$ 9 $controlecanal(identmsg) := controlecanal(identmsg) + 1$	1 $estadoprocesso(processorecv) := estadotemp$ 2 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$ 3 $canalcontrole := canalcontrole$ 4 $canallider := canallider$ 5 $historicoorigem\_votacao := historicoorigem\_votacao \cup \{\{conthistorico\_votacao \mapsto processo\}\}$ 6 $conthistorico\_votacao := conthistorico\_votacao + 1$ 7 $historicoprocdestino\_votacao := historicoprocdestino\_votacao \cup \{\{conthistorico\_votacao \mapsto processorecv\}\}$ 8 $historicoordem\_votacao := historicoordem\_votacao \cup \{\{conthistorico\_votacao \mapsto ordemprocesso\}\}$ 9 $controleordem\_votacao := controleordem\_votacao \lhd \{processorecv \mapsto (conjuntotemp \lhd \{processo \mapsto ordemprocesso\})\}$ 10 $controlecanal\_votacao(identmsg) := controlecanal\_votacao(identmsg) + 1$

Figura 4.14 – passo I5.3 e I5.4 do evento *processosrecebemmsg*

O evento *removemsg* do refinamento padrão, teve seu nome trocado na incorporação para *removemsg_votacao*. O *removemsg_votacao* foi o último evento incorporado no 2PC_2.

Para incorporação das guardas do *removemsg_votacao*, foram copiadas do refinamento padrão as guardas do evento *removemsg* (passo I5.1 da Figura 4.15). Sendo o evento *removemsg*, do refinamento padrão, um evento refinado do evento *receiver*, combiando com o evento *processos-recebemmsg*, igualmente o evento *removemsg_votacacao* é considerado o refinamento do evento *processosrecebemmsg* do 2PC_1. Para tanto foi adicionada a guarda extra do evento *processosre-*

cebemmsg nas guardas do evento *removemsg_votacao* (linha 13 do passo I5.2 da Figura 4.15).

Passo I5.1 do $removemsg\_votacao$	Passo I5.2 do $removemsg\_votacao$
1 $\{identmsg \mapsto mensagem\} \in canaldados$ 2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalprocessos$ 3 $processorecv \in grupotemp$ 4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole$ 5 $processo \in 1..NP$ 6 $processorecv \in GRUPO$ 7 $ordemprocesso \in \mathbb{N}_1$ 8 $processorecv \mapsto conjuntotemp \in controleordem$ 9 $processo \mapsto ordemtemp \in conjuntotemp$ 10 $ordemprocesso = ordemtemp + 1$ 11 $identmsg \in dom(controlecanal)$ 12 $controlecanal(identmsg) < (card(GRUPO) - 1) \wedge controlecanal(identmsg) < NP$ 13 $processorecv \in 1..NP$ 14 $conjuntotemp \in ran(controleordem)$	1 $\{identmsg \mapsto mensagem\} \in canallider$ 2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalcontrole$ 3 $processorecv \in grupotemp$ 4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole\_votacao$ 5 $processo \in 1..NP$ 6 $processorecv \in PROCESSOS$ 7 $ordemprocesso \in \mathbb{N}_1$ 8 $processorecv \mapsto conjuntotemp \in controleordem\_votacao$ 9 $processo \mapsto ordemtemp \in conjuntotemp$ 10 $ordemprocesso = ordemtemp + 1$ 11 $identmsg \in dom(controlecanal\_votacao)$ 12 $controlecanal\_votacao(identmsg) = (card(PROCESSOS) - 1)$ 13 $estadotemp \in FASEPROCESSO$ 14 $conjuntotemp \in ran(controleordem\_votacao)$ 15 $processorecv \in 1..NP$

Figura 4.15 – passo I5.1 e I5.2 do evento *removemsg_votacao*

Com exceção das guardas das linhas 3, 5, 7 e 10, as guardas presentes no passo I5.1 da Figura 4.15, precisaram ser adequadas na máquina 2PC_2. Ficando as guardas do evento *removemsg_votacao* de acordo com o passo I5.2 da Figura 4.15.

Para incorporação das ações do *removemsg_votacao*: foram adicionadas as ações do evento *removemsg* do refinamento padrão (passo 5.3 da Figura 4.16); semelhante ao ocorrido com as guardas presentes no evento *removemsg_votacao*, foi adicionado a variável extra do evento *processosrecebemmsg* nas ações do evento *removemsg_votacao* (linha 1 do passo 5.4 da Figura 4.16).

Passo I5.3 do $removemsg\_votacao$	Passo I5.4 do $removemsg\_votacao$
1 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$ 2 $canaldados := canaldados \setminus \{\{identmsg \mapsto mensagem\}\}$ 3 $canalcontrole := canalcontrole \setminus \{\{identmsg \mapsto ordemprocesso\}\}$ 4 $canalprocessos := canalprocessos \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}$ 5 $conthistorico := conthistorico + 1$ 6 $historicoorigem := historicoorigem \cup \{\{conthistorico \mapsto processo\}\}$ 7 $historicoprocdestino := historicoprocdestino \cup \{\{conthistorico \mapsto processorecv\}\}$ 8 $historicoordem := historicoordem \cup \{\{conthistorico \mapsto ordemprocesso\}\}$ 9 $controleordem := controleordem \Leftarrow \{processorecv \mapsto (conjuntotemp \Leftarrow \{processo \mapsto ordemprocesso\})\}$ 10 $controlecanal(identmsg) := controlecanal(identmsg) + 1$	1 $estadoprocesso(processorecv) := estadotemp$ 2 $dadosrecebidos := dadosrecebidos \cup \{\{identmsg \mapsto mensagem\}\}$ 3 $canalcontrole := canalcontrole \setminus \{\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\}\}$ 4 $canallider := canallider \setminus \{\{identmsg \mapsto mensagem\}\}$ 5 $canalcontrole\_votacao := canalcontrole\_votacao \setminus \{\{identmsg \mapsto ordemprocesso\}\}$ 6 $conthistorico\_votacao := conthistorico\_votacao + 1$ 7 $historicoorigem\_votacao := historicoorigem\_votacao \cup \{\{conthistorico\_votacao \mapsto processo\}\}$ 8 $historicoprocdestino\_votacao := historicoprocdestino\_votacao \cup \{\{conthistorico\_votacao \mapsto processorecv\}\}$ 9 $historicoordem\_votacao := historicoordem\_votacao \cup \{\{conthistorico\_votacao \mapsto ordemprocesso\}\}$ 10 $controleordem\_votacao := controleordem\_votacao \Leftarrow \{processorecv \mapsto (conjuntotemp \Leftarrow \{processo \mapsto ordemprocesso\})\}$ 11 $controlecanal\_votacao(identmsg) := controlecanal\_votacao(identmsg) + 1$

Figura 4.16 – passo I5.3 e I5.4 do evento *removemsg_votacao*

Nas ações do evento *remove_votacacao* apenas a linha 1 do passo I5.3, da figura 4.16, não precisou ser adequada na máquina 2PC_2. As ações do evento *remove_votacao* são representadas no passo I5.4 da Figura 4.16.

Neste ponto da modelagem a comunicação no sentido líder - participantes encontra-se ordenada,

ou seja, as determinações do líder são efetivadas ordenadamente nos processos participantes. Entretanto os processos participantes ainda não conseguem comunicar o líder de seus votos. Desta forma, foi criado um novo refinamento 2PC_3 para inserção da comunicação no sentido participantes - lider (Seção 4.3).

4.3 2PC_3

Neste passo de refinamento são introduzidas estrturas para comunicação dos processos participantes com o líder. Dada que a comunicação do sentido líder - participantes está completa, as guardas dos eventos, responsáveis por tal comunicação, recebem novas restrições no 2PC_3. Desta forma, restringindo o comportamento do modelo,ou seja, deixando mais perto da realidade do *two phase commit*.

No 2PC_3 foram adicionadas as seguintes variáveis (Figura 4.17): *canalvotacao*; *votosrecebidos*; *msgenviadasprocessos*; *canalid*; *estadovotacao*; *votodosprocessos*.

Figura 4.17 – variáveis do 2PC_3

A variável *canalvotacacao* é o canal de comunicação onde são postados os votos dos processos participantes. A variável *votosrecebidos* é o buffer onde são armazenados os votos recebidos no processo líder. A variável *msgenviadasprocessos* é o identificador da mensagem enviada. As variáveis *canalvotacao*, *votosrecebidos* e *msgenviadasprocessos* são necessárias para combinação do padrão síncrono com o modelo 2PC_3, portanto definidas pelas invariantes inv1, inv2 e inv3 de acordo a especificação padrão do unicast síncrono (Seção 3.1).

Já *canalid* representa o canal de comunicação, que indica o processo originador do voto. *Canalid* é definida como elemento de valor zero até *NP* (inv4 da Figura 4.18). A variável *estadovotacao* é utilizada para controle da votacação dos processos, sendo definida como um elemento de 2 até *NP*, mapeado para um valor booleano, ou seja, mapeado para *TRUE* ou *FALSE* (inv5 da Figura 4.18). Por fim *votodosprocessos* representa o conhecimento local do líder, sobre os votos dos processos.

votodosprocessos é definida pela inv6 da Figura 4.18, como um elemento de 2 até *NP*, mapeado para o conjunto *VOTOS*.

inv1 $canalvotacao \subseteq \mathbb{N}_1 \nrightarrow VOTOS$
inv2 $votosrecebidos \subseteq \mathbb{N}_1 \nrightarrow VOTOS$
inv3 $msgenviadasprocessos \in \mathbb{N}_1$
inv4 $canalid \in 0..NP$
inv5 $estadovotacao \in 2..NP \rightarrow BOOL$
inv6 $votodosprocessos \in 2..NP \rightarrow VOTOS$

Figura 4.18 – invariantes do 2PC_3

No refinamento 2PC_3 foram modelados 6 eventos: *initialisation*; *lidermulticast*; *processosrecebemmsg*; *removemsg_votacacao*; *processosvotacao*; *liderrecebe*.

Os eventos *initialisation*, *lidermulticast* e *processosrecebemmsg* são herdados da máquina abstrato (2PC_2). Já os eventos *processosvotacao* e *liderrecebe* foram criados neste passo de refinamento e são responsáveis pela comunicação no sentido participantes - líder.

No evento de inicialização do 2PC_3 (Figura 4.19) foram adicionadas a inicialização das variáveis criadas neste passo de refinamento. As ações das variáveis *canalvotacacao* (act15 da Figura 4.19), *votosrecebidos* (act16 da Figura 4.19) e *msgenviadasprocessos* (act17 da Figura 4.19) são necessárias para combinação, portanto sendo inicializadas de acordo com a especificação padrão do *unicast* síncrono (Seção 3.1).

A variável *canalid* é inicializada com zero, representando que não existem mensagens no canal (act18 da Figura 4.19). A variável *estadovotacao* é iniciada com todos os valores (2 até *NP*) mapeados para *FALSE* (act19 da Figura 4.19). Já *votodosprocessos*, todos os valores (2 até *NP*) foram mapeados para *semvoto* (act20 da Figura 4.19).

act1 $estadoprocesso := 2..NP \times \{initp\}$
act2 $estadolider := init$
act3 $canallider := \varnothing$
act4 $canalcontrole := \varnothing$
act5 $dadosrecebidos := \varnothing$
act6 $msgenviadas := 1$
act7 $canalcontrole\_votacao := \varnothing$
act8 $historicosend\_votacao := 1..NP \times \{0\}$
act9 $historicoorigem\_votacao := \varnothing$
act10 $conthistorico\_votacao := 0$
act11 $historicoprocdestino\_votacao := \varnothing$
act12 $historicoordem\_votacao := \varnothing$
act13 $controleordem\_votacao := 1..NP \times \{1..NP \times \{0\}\}$
act14 $controlecanal\_votacao := \mathbb{N}_1 \times \{0\}$
act15 $canalvotacao := \varnothing$
act16 $votosrecebidos := \varnothing$
act17 $msgenviadasprocessos := 1$
act18 $canalid := 0$
act19 $estadovotacao := 2..NP \times \{FALSE\}$
act20 $votodosprocessos := 2..NP \times \{semvoto\}$

Figura 4.19 – evento de inicialização do 2PC_3

No evento *multicastlider* foi adicionado uma guarda, afim restringir a troca de estado do processo líder e ordenadar o tipo de mensagens a ser enviada levando em conta a fase do líder no *two phase commit*. Para tanto, na guarda grd6 da Figura 4.20, determinou-se que: se o estado do líder for *init*

então a mensagem a ser enviada, para os participantes, deve ser *vote_request* e o próximo estado do líder *wait*.

grd1 $mensagem \in MSGLIDER$
grd2 $processoenvia \in 1..NP$
grd3 $processoenvia = LIDER$
grd4 $msgenviadas \in \mathbb{N}_1$
grd5 $estadotemp \in FASELIDER$
grd6 $estadolider = init \Rightarrow estadotemp = wait \wedge mensagem = vote\_request$

Figura 4.20 – guardas do evento *multicastlider* 2PC_3

Da mesma forma, nos eventos *processosrecebemmsg* e *removemsg_votacao* foram adicionadas guardas para restringir o comportamento da troca de fase dos processos participantes. As guardas adicionada nos eventos determina que: se o estado do processo participante na recepção, for *initp* e a mensagem no canal for *vote_resquet*, então o participante receptor muda seu estado para *ready*.

processosrecebemmsg	*removemsg_votacao*
grd1 $\{identmsg \mapsto mensagem\} \in canallider$ grd2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalcontrole$ grd3 $processorecv \in grupotemp$ grd4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole\_votacao$ grd5 $processo \in 1..NP$ grd6 $processorecv \in PROCESSOS$ grd7 $ordemprocesso \in \mathbb{N}_1$ grd8 $processorecv \mapsto conjuntotemp \in controleordem\_votacao$ grd9 $processo \mapsto ordemtemp \in conjuntotemp$ grd10 $ordemprocesso = ordemtemp + 1$ grd11 $identmsg \in dom(controlecanal\_votacao)$ grd12 $controlecanal\_votacao(identmsg) < (card(PROCESSOS) - 1) \wedge controlecanal\_votacao(identmsg) < NP$ grd13 $processorecv \in 1..NP$ grd14 $conjuntotemp \in ran(controleordem\_votacao)$ grd15 $estadotemp \in FASEPROCESSO$ grd16 $processorecv \in dom(estadoprocesso) \wedge estadoprocesso(processorecv) = initp \wedge mensagem = vote\_request \Rightarrow estadotemp = ready$	grd1 $\{identmsg \mapsto mensagem\} \in canallider$ grd2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalcontrole$ grd3 $processorecv \in grupotemp$ grd4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole\_votacao$ grd5 $processo \in 1..NP$ grd6 $processorecv \in PROCESSOS$ grd7 $ordemprocesso \in \mathbb{N}_1$ grd8 $processorecv \mapsto conjuntotemp \in controleordem\_votacao$ grd9 $processo \mapsto ordemtemp \in conjuntotemp$ grd10 $ordemprocesso = ordemtemp + 1$ grd11 $identmsg \in dom(controlecanal\_votacao)$ grd12 $controlecanal\_votacao(identmsg) = (card(PROCESSOS) - 1)$ grd13 $estadotemp \in FASEPROCESSO$ grd14 $conjuntotemp \in ran(controleordem\_votacao)$ grd15 $processorecv \in 1..NP$ grd16 $processorecv \in dom(estadoprocesso) \wedge estadoprocesso(processorecv) = initp \wedge mensagem = vote\_request \Rightarrow estadotemp = ready$

Figura 4.21 – guardas dos eventos *processosrecebemmsg* e *removemsg_votacao* 2PC_3

O evento *processosvotacao*, criado neste refinamento, é responsável pelo envio das mensagens com os votos dos processos participantes para o líder.

Para evento *processosvotacao* ocorrer, as seguintes guardas deverão ser verdadeiras: o voto dos processo participantes devem pertencer ao conjunto *VOTO* (grd1 da Figura 4.22); o identificador associado a mensagem tem quer ser um valor natural (grd2 da Figura 4.22); o processo originador tem que fazer parte dos processos participantes (grd3 da Figura 4.22); o processo originador tem que estar no estado *ready* (grd4 da Figura 4.22); o processo originador não pode ter votado (grd5 da Figura 4.22); o voto tem quer ser diferente de *semvoto* (grd6 da Figura 4.22).

As guardas gr1 e grd2 da Figura 4.22 são derivadas do evento *sender* da especificação padrão do *unicast* síncrono (Seção 3.1), ou seja, são necessárias para combinação do padrão.

Sendo verdadeiras as guardas do evento *processovotacao* as seguintes ações ocorrem: o canal

grd1 $voto \in VOTOS$
grd2 $msgenviadasprocessos \in \mathbb{N}_1$
grd3 $processo \in 2..NP$
grd4 $processo \in dom(estadoprocesso) \wedge estadoprocesso(processo) = ready$
grd5 $estadovotacao(processo) = FALSE$
grd6 $voto \neq semvoto$

Figura 4.22 – Guardas do evento *processovotacao* 2PC_3

de comunicação *canalvotacao* recebe o voto de um processo participante, assoaciado a uma identificação (act1 da Figura 4.23); *msgenviadasprocessos* é incrementando em uma unidade (act2 da Figura 4.23); canalid recebe o PID (identificação) do processo participante originador do voto(act3 da Figura 4.23); a realização do voto do processo é registrado para que ele não posso votar novamente (act4 da Figura 4.23);

act1 $canalvotacao := canalvotacao \cup \{\{msgenviadasprocessos \mapsto voto\}\}$
act2 $msgenviadasprocessos := msgenviadasprocessos + 1$
act3 $canalid := processo$
act4 $estadovotacao(processo) := TRUE$

Figura 4.23 – ações do evento *processovotacao* 2PC_3

As ações act1 e act2 são derivadas do evento *sender* da especificação, padrão do *unicast* síncrono, sendo necessárias para combinação do padrão.

Para o evento *liderrecebe* acontecer as seguintes guardas deverão ser verdadeiras: deve existir um voto no canal de comunicação *canalvotacao* (grd1 da Figura 4.24); deve existir um PID no *canalid* (grd2 da Figura 4.24); o PID no canal deve ser de um processo participante (grd3 da Figura 4.24); o voto do canal tem que ser um voto válido (grd4 da Figura 4.24);

grd1 $\{identmsg \mapsto voto\} \in canalvotacao$
grd2 $pid = canalid$
grd3 $pid \in 2..NP$
grd4 $voto \neq semvoto$

Figura 4.24 – Guardas do evento *liderrecebe* 2PC_3

A guarda grd1 é derivada do evento *receiver* da especificação padrão, do padrão *unicast* síncrono, portanto sendo utilizada para combinação do padrão.

Sendo verdadeiras as guarda, as seguintes ações são realizadas no evento *liderrecebe*: o buffer recebe o voto do processo participante, presente no canal de comunicação (act1 da Figura 4.25); uma ação não determinística é realizada no *canalvotacao*, determinando que o voto pode permanecer no canal ou ser removido (act2 da Figura 4.25); o voto do processo participantes é computado pelo líder (act3 da Figura 4.25); o *canalid* é limpo (act4 da Figura 4.25);

act1 $votosrecebidos := votosrecebidos \cup \{\{identmsg \mapsto voto\}\}$
act2 $canalvotacao :\in \{canalvotacao, canalvotacao \setminus \{\{identmsg \mapsto voto\}\}\}$
act3 $votodosprocessos(pid) := voto$
act4 $canalid := 0$

Figura 4.25 – Ações do evento *liderrecebe* 2PC_3

Terminada a modelagem do 2PC_3, percebe-se que o refinamento gerado introduz um canal simples de comunicação e dois novos eventos: um responsável pelo envio dos votos dos participantes e outro pela recepção dos votos pelo líder.

As guardas dos eventos, responsáveis pela comunicação do líder para os processos participantes, foram restringidas, limitando as trocas de estados iniciais, para um modo ordenado no processo líder e nos processos participantes.

As estruturas introduzidas, neste refinamento, propiciam a combinação com o padrão *unicast* síncrono, dado que neste nível da modelagem ainda não existe ordem na comunicação participante - líder.

A ausencia de ordem no sentido participante - líder acarreta na repetição ou perca de votos, além da impossibilidade de restringir o evento líder para uma tomada de decisão (*abort* ou *commit*);

Para tanto é necessário a incorporação do padrão *unicast* síncrono. Para incorporação do padrão, novamente foram realizados três fases para garantir a corretude da incorporação: combinação, verificação e incorporação.

4.3.1 Combinação

As variáveis combinadas (passo C1), na incorporação do *unicast* síncrono, foram : *canaldados* da especificação padrão (Seção 3.1) com *canalvotacao* do 2PC_3 (linha 1 Tabela 4.4); *dadorecebidos* da especificação padrão (Seção 3.1) com *votosrecebidos* do 2PC_3 (linha 2 Tabela 4.4); *msgenviadas* da especificação padrão (Seção 3.1) com *msgenviadasprocessos* do 2PC_3 (linha 3 Tabela 4.4).

As variáveis extras presentes na combinação são apresentadas da linha 4 até a linha 20 da Tabela 4.4.

Tabela 4.4 – Passo C1

	Variáveis Combinadas ($\rightsquigarrow$)	
Linha	Especificação padrão	2PC_3
1	$canaldados$	$canalvotacao$
2	$dadosrecebidos$	$votosrecebidos$
3	$msgenviadas$	$msgenviadasprocessos$
	Variáveis não combinadas (extras)	
4	$estadoprocesso$	
5	$estadolider$	
6	$canallider$	
7	$canalcontrole$	
8	$dadosrecebidos$	
9	$msgenviadas$	
10	$canalcontrole\_votacao$	
11	$historicosend\_votacao$	
12	$historicoorigem\_votacao$	
13	$conthistorico\_votacao$	
14	$historicoprocdestino\_votacao$	
15	$historicoordem\_votacao$	
16	$controleordem\_votacao$	
17	$controlecanal\_votacao$	
18	$canalid$	
19	$estadovotacao$	
20	$votodosprocessos$	

No passo C2, foram definidos os seguintes eventos combinados: *initialisation* da especificação padrão (Seção 3.1) com *initialisation* do 2PC_3 (linha 1 da Tabela 4.4); *sender* da especificação

padrão (Seção 3.1) com *processosvotacao* do 2PC_3 (linha 2 da Tabela 4.4); *receiver* da especificação padrão (Seção 3.1) com *liderrecebe* do 2PC_3 (linha 3 da Tabela 4.4).

Da linha 4 até a linha 7 da Tabela 4.4 são apresentados os eventos extras da combinação.

Tabela 4.5 – Passo C2

	Eventos Combinados ($\rightsquigarrow$)	
Linha	Especificação padrão	2PC_3
1	*initialisation*	*initialisation*
2	*sender*	*processosvotacao*
3	*receiver*	*liderrecebe*
4	Eventos não Combinados (Extras)	
5	*lidermulticast*	
6	*processosrecebemmsg*	
7	*removemsg_votacao*	

Definida as combinações, os eventos e variáveis extras, realiza-se a verificação das combinações.

4.3.2 Verificação

Realizando o passo V3 da verificação, para os eventos *initialisation* da especificação padrão e do 2PC_3, percebe-se que as ações dos eventos são sintaticamente iguais. Já no passo V4, do evento *initialisation* (2PC_3), foram encontradas as seguintes ações extras: act1, act2, act3, act4, act5, act6, act7, act8, act9, act10, act11, act12, act13, act14, act18, act19, act20 da Figura 4.19.

No passo V1 e V2 da verificação dos eventos *sender* e *processosvotacao*, nota-se que as guardas são sintaticamente iguais e que o evento *processosvotacao* contém as seguintes guardas extras: grd3, grd4, grd5, grd6 da Figura 4.22.

Assim como as guardas, no passo V3, foi verificado que as ações dos eventos *sender* e *processosvotacao* são sintaticamente iguais e que o evento *processosvotacao* contém as ações act3 e act4 da Figura 4.23, como ações extras.

Na verificação dos últimos eventos combinados *receiver* e *liderrecebe*, as guardas e ações foram consideradas sintaticamente iguais (passo V1 e V3). Além disso, no passo V2, foram encontradas as guardas extras grd2, grd3 e grd4 da Figura 4.24. No passo V4 foram encontradas as ações extras act3 e act4 da Figura 4.25.

Tendo como correta todas as verificações, anteriomente citadas, para a incorporação foi gerado um novo refinamento chamado 2PC_4 (Seção 4.4)

4.4 2PC_4

No refinamento 2PC_4 é incorporado o padrão *unicast* síncrono aos eventos combinados no refinamento anterior (2PC_3).

No passo I1 da incorporação, foram copiadas as variáveis do refinamento padrão, do padrão *unicast* síncrono (Seção 3.2), para o refinamento 2PC_4. Semelhante a incorporação da Seção 4.2, foram realizada as trocas necessárias nas variáveis da máquina. As variáveis presentes no 2PC_4 podem ser visualizadas na Figura 4.26.

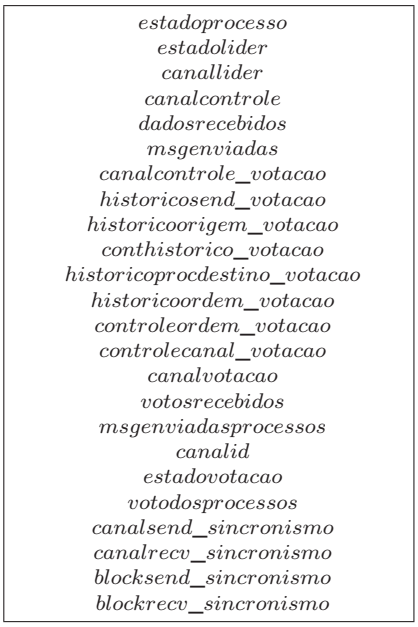


Figura 4.26 – variáveis do 2PC_4

Analisando a Figura 4.26 é possivel perceber que, neste refinamento, foi adicionado o sufixo *_sincronismo* nas 4 variáveis oriundas do refinamento padrão.

Após a incorporação das variáveis no 2PC_4, foi incorporado as invariantes (passo I2). Para tanto, foram copiadas as invariantes da especificação padrão (Seção 3.1) e do refinamento padrão, do *unicast* síncrono (Seção 3.2). A seguir, foram realizadas as trocas necessárias para adequação das invariantes no refinamento, com isso resultando nas invariantes apresentadas na Figura 4.27.

No passo I3, foram incorporados os eventos *multicastlider*, *processosrecebemmsg* e *removemsg _votacao*, ou seja, os eventos não combinados. Para incorporação, deste tipo de evento, foi copiado os eventos sem nenhum modificação nas guardas ou ações. Portanto os eventos *multicastlider*, *processosrecebemmsg* e *removemsg_votacao*, do refinamento do 2PC_4, tratam-se dos mesmo eventos descritos anteriormente no 2PC_3.

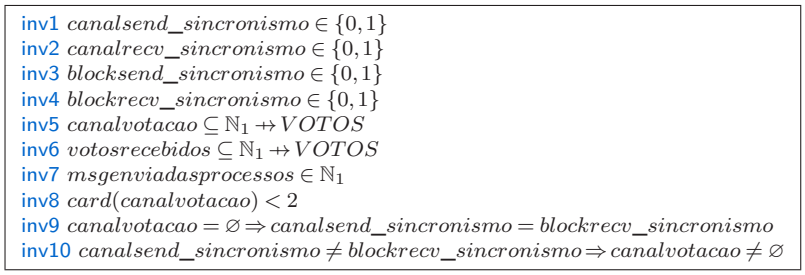


Figura 4.27 – Invariantes do 2PC_4

A incorporação dos eventos combiandos, foi realizada evento a evento. O primeiro evento incorporado foi o *initialisation*. Para este evento foram copiadas as ações do evento *initialisation* do refinamento padrão (Seção 3.2) e adicionadas as variáveis extras do evento *initialisation* do 2PC_3. Por fim realizado as mudanças nas ações do evento, afim de adequar as mesmas ao refinamento 2PC_4.

As ações do evento de inicialização são apresentadas na Figura 4.28.

act1 $estadoprocesso := 2..NP \times \{initp\}$
act2 $estadolider := init$
act3 $canallider := \varnothing$
act4 $canalcontrole := \varnothing$
act5 $dadosrecebidos := \varnothing$
act6 $msgenviadas := 1$
act7 $canalcontrole\_votacao := \varnothing$
act8 $historicosend\_votacao := 1..NP \times \{0\}$
act9 $historicoorigem\_votacao := \varnothing$
act10 $conthistorico\_votacao := 0$
act11 $historicoprocdestino\_votacao := \varnothing$
act12 $historicoordem\_votacao := \varnothing$
act13 $controleordem\_votacao := 1..NP \times \{1..NP \times \{0\}\}$
act14 $controlecanal\_votacao := \mathbb{N}_1 \times \{0\}$
act15 $canalvotacao := \varnothing$
act16 $votosrecebidos := \varnothing$
act17 $msgenviadasprocessos := 1$
act18 $canalid := 0$
act19 $estadovotacao := 2..NP \times \{FALSE\}$
act20 $votodosprocessos := 2..NP \times \{semvoto\}$
act21 $canalsend\_sincronismo := 0$
act22 $canalrecv\_sincronismo := 0$
act23 $blocksend\_sincronismo := 0$
act24 $blockrecv\_sincronismo := 0$

Figura 4.28 – Evento de inicialização do 2PC_4

No evento *processovotacao*, inicialmente foi incorporado sua guarda ao refinamento do 2PC_4. Para incorporação da guardas, foram copiadas as guardas do evento *sender*, do refinamento padrão, do padrão *unicast* síncrono (Seção 3.2) e adicionadas as guardas extras encontradas no passo V2 do evento *processovotacao*. Logo após foi realizado as modificações necessárias nas guardas, para adequar as mesmas ao refinamento 2PC_4. Desta forma, ficando as guardas do evento *processovotacao* de acordo com a Figura 4.29.

grd1 $voto \in VOTOS$
grd2 $msgenviadasprocessos \in \mathbb{N}_1$
grd3 $processo \in 2..NP$
grd4 $processo \in dom(estadoprocesso) \land estadoprocesso(processo) = ready$
grd5 $estadovotacao(processo) = FALSE$
grd6 $voto \neq semvoto$
grd7 $canalrecv\_sincronismo = blocksend\_sincronismo$
grd8 $canalvotacao = \varnothing$

Figura 4.29 – Guardas do evento *processovotacao* do 2PC_4

Para incorporação das ações, do evento *processovotacao*, foram copiadas as ações do evento *sender* do refinamento padrão, do padrão *unicast* síncrono e adicionadas as ações extras do *processovotacao*. Igualmente as guardas, as ações foram adaptadas a realidade do 2PC_4. As ações do *processovotacao* no 2PC_4 podem ser visualizada na Figura 4.30.

No evento *liderrecebe*, inicialmente foram incorporadas as guardas, ou seja, copiaram-se as guardas do evento *receiver* do refinamento padrão, do padrão *unicast* síncrono, depois adicionaram-se as guardas extras. Por fim realizaram as modificações necessárias nas guardas para adequar, as mesmas, ao refinamento 2PC_4. A Figura 4.31 apresenta as guardas do evento *liderrecebe* no 2PC_4.

act1 $canalvotacao := canalvotacao \cup \{\{msgenviadasprocessos \mapsto voto\}\}$
act2 $msgenviadasprocessos := msgenviadasprocessos + 1$
act3 $canalid := processo$
act4 $estadovotacao(processo) := TRUE$
act5 $canalsend\_sincronismo := 1 - blocksend\_sincronismo$
act6 $blocksend\_sincronismo := 1 - blocksend\_sincronismo$

Figura 4.30 – Ações do evento *processovotacao* do 2PC_4

grd1 $\{identmsg \mapsto voto\} \in canalvotacao$
grd2 $pid = canalid$
grd3 $pid \in 2..NP$
grd4 $voto \neq semvoto$
grd5 $canalsend\_sincronismo \neq blockrecv\_sincronismo$

Figura 4.31 – Guardas do evento *liderrecebe* do 2PC_4

Para incorporação das ações evento *liderrecebe* foram realizadas as seguintes ações: adicionadas as guardas do evento *receiver* do refinamento padrão, do padrão unicast síncrono; adicionadas as ações extras do evento *liderrecebe*; realizadas as modificações necessárias nas ações.

As ações do evento *liderrecebe* no refinamento 2PC_4 podem ser visualizadas nas Figura 4.32.

act1 $votosrecebidos := votosrecebidos \cup \{\{identmsg \mapsto voto\}\}$
act2 $canalvotacao := canalvotacao \setminus \{\{identmsg \mapsto voto\}\}$
act3 $votodosprocessos(pid) := voto$
act4 $canalid := 0$
act5 $canalrecv\_sincronismo := 1 - blockrecv\_sincronismo$
act6 $blockrecv\_sincronismo := 1 - blockrecv\_sincronismo$

Figura 4.32 – Ações do evento $liderrecebe$ do 2PC_4

Terminada a incorporação do padrão unicast síncrono no *two phase commit*, nota-se que, no canal introduzido no refinamento anterior (2PC_3), a votação dos processos participantes para o líder passou a ser realizada de modo síncrono.

Assim, quando um participante está votando, outro processo não pode realizar a votação até que o líder seja desbloqueado. Neste momento o modelo contém comunicação ordenada nos dois sentidos. O líder se comunica, através de mensagens *multicast* assíncronas de ordem FIFO, com os processos participantes e os processos participantes mandam seus votos de modo unicast síncrono para o líder.

A partir do 2PC_4 foi gerado um novo refinamento para adicionar as características de decisão do líder.

4.5 2PC_5

Finalmente, com toda a estrutura de canais de comunicação, pode-se refinar a máquina anterior e introduzir a decisão do coordenador com base nos votos dos participantes.

Para decisão do coordenador, foram adicionadas novas guardas no evento *lidermulticast* que determinam que: se o estado do líder for *wait* e todos os processos participantes votarem por *commit*, então o líder muda seu estado para *commit* e envia a mensagem de *vote_commit*, para os participantes (grd7 da Figura 4.33); se o estado do líder for *wait* e exitir um processo que votou por

abort, então o líder muda seu estado para *abort* e manda uma mensagem de *vote_abort*, para os processos participantes (grd8 da Figura 4.33); o estado do líder tem que ser diferente de *commit* e *abort*, para enviar uma mensagem para os processos participantes (grd9 da Figura 4.33); se o estado do líder for *wait* o líder somente poderá mudar seus estado para *commit* ou *abort* (grd10 da Figura 4.33);

grd1 $mensagem \in MSGLIDER$
grd2 $processoenvia \in 1..NP$
grd3 $processoenvia = LIDER$
grd4 $msgenviadas \in \mathbb{N}_1$
grd5 $estadotemp \in FASELIDER$
grd6 $estadolider = init \Rightarrow estadotemp = wait \wedge mensagem = vote\_request$
grd7 $estadolider = wait \wedge votodosprocessos[2..NP] = \{processocommit\} \Rightarrow (estadotemp = commit \wedge mensagem = vote\_commit)$
grd8 $estadolider = wait \wedge (\exists x \cdot (x \in dom(votodosprocessos) \wedge votodosprocessos(x) = processoabort))$
$\Rightarrow (estadotemp = abort \wedge mensagem = vote\_abort)$
grd9 $estadolider \neq commit \wedge estadolider \neq abort$
grd10 $estadolider = wait \Rightarrow ((votodosprocessos[2..NP] = \{processocommit\}) \vee$
$(\exists x \cdot (x \in dom(votodosprocessos) \wedge votodosprocessos(x) = processoabort)))$

Figura 4.33 – Guardas do evento *lidermulticast* do 2PC_5

Os processos participantes devem respeitar a decisão do líder e mudar seu estado para *abort* ou *commit*. Para tanto, foi adicionado restrições nas guardas dos eventos de recepção das mensagens enviadas pelo líder, de modo que, os processos participantes realizem a decisão do líder.

Foram adicionadas as seguintes restrições nas guardas dos eventos *processosrecebemmsg* e *removemsg_votacao*: se o estado do processo participante for *ready* e receber uma mensagem de *vote_commit* do líder, o processo participante muda seu estado para *commitp* (grd17 dos eventos da Figura 4.34); se o estado do processo participante for *ready* e receber uma mensagem de *vote_abort* do líder, o processo participante muda seu estado para *abortp* (grd18 dos eventos da Figura 4.34).

Dada a modelagem do *two phase commit*, nota-se que até o terceiro refinamento, como não existia uma comunicação confiável entre os processos participantes e o líder, não era possível garantir que o líder chegava a uma decisão. Com isso era inviável introduzir estruturas comportamentais e provas relacionadas as possíveis decisões do líder no modelo (abort ou commit), dado que a decisão do líder é baseada nos votos dos processos participantes.

No quarto refinamento, entretando, foi possível introduzir as caracerísticas comportamentais da votação dos processos no modelo.

processorecebemmsg	*removemsg_votacao*
grd1 $\{identmsg \mapsto mensagem\} \in canallider$ grd2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalcontrole$ grd3 $processorecv \in grupotemp$ grd4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole\_votacao$ grd5 $processo \in 1..NP$ grd6 $processorecv \in PROCESSOS$ grd7 $ordemprocesso \in \mathbb{N}_1$ grd8 $processorecv \mapsto conjuntotemp \in controleordem\_votacao$ grd9 $processo \mapsto ordemtemp \in conjuntotemp$ grd10 $ordemprocesso = ordemtemp + 1$ grd11 $identmsg \in dom(controlecanal\_votacao)$ grd12 $controlecanal\_votacao(identmsg) < (card(PROCESSOS) - 1) \wedge controlecanal\_votacao(identmsg) < NP$ grd13 $processorecv \in 1..NP$ grd14 $conjuntotemp \in ran(controleordem\_votacao)$ grd15 $estadotemp \in FASEPROCESSO$ grd16 $processorecv \in dom(estadoprocesso) \wedge estadoprocesso(processorecv) = initp \wedge mensagem = vote\_request \Rightarrow estadotemp = ready$ grd17 $estadoprocesso(processorecv) = ready \wedge mensagem = vote\_commit \Rightarrow estadotemp = commitp$ grd18 $estadoprocesso(processorecv) = ready \wedge mensagem = vote\_abort \Rightarrow estadotemp = abortp$	grd1 $\{identmsg \mapsto mensagem\} \in canallider$ grd2 $\{identmsg \mapsto \{(processo \mapsto grupotemp)\}\} \in canalcontrole$ grd3 $processorecv \in grupotemp$ grd4 $\{identmsg \mapsto ordemprocesso\} \in canalcontrole\_votacao$ grd5 $processo \in 1..NP$ grd6 $processorecv \in PROCESSOS$ grd7 $ordemprocesso \in \mathbb{N}_1$ grd8 $processorecv \mapsto conjuntotemp \in controleordem\_votacao$ grd9 $processo \mapsto ordemtemp \in conjuntotemp$ grd10 $ordemprocesso = ordemtemp + 1$ grd11 $identmsg \in dom(controlecanal\_votacao)$ grd12 $controlecanal\_votacao(identmsg) = (card(PROCESSOS) - 1)$ grd13 $estadotemp \in FASEPROCESSO$ grd14 $conjuntotemp \in ran(controleordem\_votacao)$ grd15 $processorecv \in 1..NP$ grd16 $processorecv \in dom(estadoprocesso) \wedge estadoprocesso(processorecv) = initp \wedge mensagem = vote\_request \Rightarrow estadotemp = ready$ grd17 $estadoprocesso(processorecv) = ready \wedge mensagem = vote\_commit \Rightarrow estadotemp = commitp$ grd18 $estadoprocesso(processorecv) = ready \wedge mensagem = vote\_abort \Rightarrow estadotemp = abortp$

Figura 4.34 – Guardas do evento $processosrecebemmsg$ do 2PC_5

5. CONCLUSÃO

Este trabalho abordou a utilização do formalismo Event-B, para especificação de sistemas distribuídos, de forma a garantir, já em fases iniciais de modelagem, que as propriedades desejadas do sistema sejam preservadas.

Na realização da modelagem dos sistemas distribuídos em Event-B, notou-se a possibilidade de classificar as diversas semânticas de comunicação conforme restrições do não determinísmo, nas possíveis computações permitidas, em termos de troca de mensagens. Desta forma, viabilizou-se a construção da biblioteca de padrões de comunicação para mecanismo de trocas de mensagens. A biblioteca permite o reuso das semânticas ,e portanto, o reuso das provas.

Com isso, este trabalho se diferencia dos citados na Seção 2.4, por não se tratar de um sistema específico modelado em Event-B, mas de uma biblioteca de padrões de especificação, que suportam diversas semânticas de comunicação por troca de mensagem. Portanto a contribuição aqui feita apresenta um conjunto abrangente de padrões para diversas semânticas de comunicação possíveis.

A aplicação dos padrões da biblioteca é facilitada, dado o fator que os padrões são modelados em dois níveis: modelo abstrato (especificação padrão) usado para combinação com o modelo alvo; modelo concreto (refinamento padrão) usado para incorporar as características distribuídas no modelo alvo.

Os padrões contidos na biblioteca de padrões, livram o desenvolvedor de se preocupar com a comunicação na modelagem de sistemas distribuídos em Event-B. Fato que foi demostrado neste trabalho, através da aplicação de dois padrões da biblioteca em um sistema distribuído o *two phase commit*.

Para aplicação do padrão, apenas foi necessário se preocupar com as estruturas mínimas necessárias para combinação. Além disso, o conceito de refinamento, existente em Event-B, permitiu que os padrões fossem aplicados em passos de refinamento, portanto a cada passo agregando canais em diferentes sentidos, como novas características, tanto de comunicação através da incorporação do padrão, como características de comportamento do *two phase commit*.

Para modelagem dos sistemas distribuídos e dos padrões da biblioteca, em Event-B, foi utilizado a plataforma RODIN. A plataforma proporcionou uma interação amigável com o processo de provas inerente ao método. Na modelagem dos sistemas, grande parte das obrigações de provas, necessárias para garantir a corretude dos sistemas em Event-B, eram descartadas automaticamente, dando um rápido *feed back* sobre a corretude dos sistemas. Entretanto, alguns casos foram necessárias a interação para realização das provas. No entanto, percebeu-se que na contrução de propriedades mais elaboras ou grandes quantidades de características adiciondas em um nível, algumas provas não são satisfeitas pela ferramenta. Desta forma, tornando-se árdua a utilização dos teoremas para usuários sem conhecimentos avançados sobre as obrigações de prova e estratégias de prova dos provadores.

Como trabalhos futuros, pretende-se:

- adicionar nos padrões da biblioteca mais propriedades representando o comportamento das semânticas de comunicação modeladas.
- introduzir novos padrões, na biblioteca de padrões, de forma a abranger uma maior quantidade de semânticas de comunicação.
- criar uma biblioteca com especificação de mecanismos de tolerância a falhas modelados em Event-B.
- representar falhas em sistemas distribuídos em Event-B.

REFERÊNCIAS BIBLIOGRÁFICAS

[Abr01] J. R. Abrial. "Event Driven Distributed Program construction". Capturado em: http://www.atelierb.eu/ressources/articles/dis.pdf, Dezembro 2008.

[AH07] J. R. Abrial, S. Hallerstede. "Refinement, Decomposition, and Instantiation of Discrete Models: Application to event-b", *Fundamenta Informaticae*, vol. 77, 2007, pp. 1 -28.

[AM05] J. R. Abrial, C. Métayer. "Rodin Deliverable 3.2 - Event-B language", Technical report,Newcastle University, England, 2005.

[BH07] M. Butler, S. Hallerstede. "The Rodin Formal Modelling Tool", BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, Dezembro 2007.

[CM06] D. Cansell, D. Méry. "Formal and Incremental Construction of Distributed Algorithms: on the Distributed Reference Counting Algorithm", *Theoretical Computer Science*, vol. 364-3, 2006, pp. 318 -337.

[DKC05] J. Dollimore, T. Kindberg, G. Coulouris. "Distributed Systems: Concepts and Design (International Computer Science Series)". Addison Wesley, 2005, 944p.

[FS00] M. Fowler and K. Scoot. "UML Essencial". Bookman, 2000, pp 19 -26.

[Für09] A. Fürst. "Design Patterns in Event-B and Their Tool Support". Dissertação de Mestrado, Information Security Group Chair of Information Security Department of Computer Science, 2009, 111p.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison Wesley,1995, 416p.

[GR92] J. Gray, A. Reuter. "Transaction Processing : Concepts and Techniques". Morgan Kaufmann, 1992, 1070p.

[Hal06] S. Hallerstede. "Justifications for The Event-B Modelling Notation", *B 2007: Formal Specification and Development in B*, vol. 4355, 2006, pp 49 -63.

[Hol97] G. J. Holzmann. "The Model Checker SPIN", *IEEE Transactions on Software Engineering*, vol. 23-5 , Maio 1997, 279 -295.

[Ili07] A. Iliasov. "Refinement Patterns for Rapid Development of Dependable Systems". In: EFTS 07: Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems, 2007, 10p.

[IR08] A. Iliasov, A. Romanovsky. "Refinement Patterns for Fault Tolerant Systems", *The*

Seventh European Dependable Computing Conference (EDCC-7), vol. 8, May 2008, pp 167 -176.

[Lam78] L. Lamport. "Time, Clocks, and The Ordering of Events in a Distributed System", *Commun*. ACM, vol. 21-7, julho 1978, pp. 558 -565.

[LT89] N. Lynch, M. Tuttle. "An Introduction to Input/Output Automata", *CWI-Quarterly*, Vol. 2-3, 1989, pp. 219 -246.

[MAV05] C. Métayer, J. R. Abrial, L. Voisin. "Event-B Language". Capturado em: http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, abril 2008.

[MD01] L. Moreau, J. Duprat. "A construction of Distributed Reference Counting", *Acta Informatica*, vol. 37-8, 2001, pp 563 -595.

[Mil99] R. Milner. "Communicating and Mobile Systems: The π - Calculus". Cambridge University Press, 1999, 174p.

[Sch01] S. Schneider. "The B-method an Instruction", Palgrave Macmillan, 2001, 384p.

[TK99] H. Tipton, M. Krause. "Information Security Management Handbook". Auerbach Publications, 1999, 2036p.

[TS07] A. S. Tanenbaum, M. V. Steen. "Sistemas Distribuídos Princípios e Paradigmas". Prentice Hall, 2007, 416p.

[Tv02] A. S. Tanenbaum and M. v. Steen. "Distributed Systems: Principles and Paradigms". Prentice Hall, 2002, 803p.

[YA07] A. B. Younes , L. J. B. Ayed. "Using UML Activity Diagrams and Event-B for Distributed and Parallel Applications". In: COMPSAC 07: Proceedings of the 31st Annual International Computer Software and Applications Conference, 2007, pp. 163 -170.

[YB06] D. Yadav, M. Butler. "Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems Using Event-B". *Rigorous Development of Complex Fault-Tolerant Systems*, 2006, pp 343 -363

[ZRR02] A. F. Zorzo, B. Randell, A. Romanovsky. "Concurrency in Dependable Computing", Kluwer Academic Publishers, 2002, pp 41 -59

A. Sistemas Distribuídos Modelados em Event-B

Neste apêndice são apresentados três sistemas distribuídos modelados em event-B: to from/command apresentado em [ZRR02] (Seção A.1); algoritmo de exclusão mútua centralizado (Seção A.2); algoritmo de comunicação em grupo (Seção A.3).

A.1 To From/Command

Nesta seção são apresentados o contexto (Seção A.1.1) e a máquina (Seção A.1.2) do to from/command, modelado em Event-B. O to from/commando permite tanto envio, como a recepção de modo síncrono e assíncrono. Na realização do envio, primeiramente a mensagem é entregue a *buffer*. Da *buffer* a mensagem é entregue ao processo *receiver*.

No envio de uma mensagem, em modo síncrono, o *sender* fica bloqueado até o *receiver* informar a recepção da mensagem. Já no envio assíncrono o sender permanece desbloqueado.

Na recepção de uma mensagem, em modo síncrono, o *receiver* fica bloqueado até ser enviada uma mensagem pelo *sender*. Já na recepção assíncrona, o *receiver* permanece desbloqueado, retornando valor vazio, representando ausência de conteúdo para receber.

Além do evento de inicialização, este modelo, contém os seguintes eventos: *send*, *bufrecebe*, *bufrecebe2*, *bufretira*, *recv*, *bufretira2*, *sendblock*, *desblocksend*, *recvblock*.

O evento *send* é reponsável por enviar as mensagens assíncronas para *buffer*. O Evento *bufrecebe* é responsável por salvar as mensagens até o penúltimo endereço da *buffer*. O evento *bufrecebe2* é responsável por salvar as mensagems no último endereço da *buffer*, sinalizando buffer cheia. O evento *bufretira* envia as mensagens, salvas na *buffer*, para o *receiver* até o penúltimo endereço da *buffer* e *bufretira2* no último endereço. O evento *sendblock* envia mensagens síncronas, bloqueando o *sender*. O evento *desblocksend* desbloqueia o *sender*, na ocorrência da recepção de uma mensagem síncrona, pelo *receiver*. O evento *recv* realiza a recepção de uma mensagem de modo assíncrono e o evento *recvblock* de modo síncrono.

A.1.1 Contexto

An Event-B Specification of contexto1
Generated Date: 26 Jan 2010 @ 05:40:48 PM

CONTEXT contexto1

SETS

dados

CONSTANTS

tamanho

d1

d2

d3

d4

d5

d6

d7

d8

d9

d10

vazio

AXIOMS

axm1 : $tamanho = 3$

axm3 : $dados = \{vazio, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10\}$

axm4 : $vazio \neq d1 \wedge vazio \neq d2 \wedge vazio \neq d3 \wedge vazio \neq d4 \wedge vazio \neq d5 \wedge vazio \neq d6 \wedge vazio \neq d7 \wedge vazio \neq d8 \wedge vazio \neq d9 \wedge vazio \neq d10$

axm5 : $d1 \neq d2 \wedge d1 \neq d3 \wedge d1 \neq d4 \wedge d1 \neq d5 \wedge d1 \neq d6 \wedge d1 \neq d7 \wedge d1 \neq d8 \wedge d1 \neq d9 \wedge d1 \neq d10$

axm6 : $d2 \neq d3 \wedge d2 \neq d4 \wedge d2 \neq d5 \wedge d2 \neq d6 \wedge d2 \neq d7 \wedge d2 \neq d8 \wedge d2 \neq d9 \wedge d2 \neq d10$

axm7 : $d3 \neq d4 \wedge d3 \neq d5 \wedge d3 \neq d6 \wedge d3 \neq d7 \wedge d3 \neq d8 \wedge d3 \neq d9 \wedge d3 \neq d10$

axm8 : $d4 \neq d5 \wedge d4 \neq d6 \wedge d4 \neq d7 \wedge d4 \neq d8 \wedge d4 \neq d9 \wedge d4 \neq d10$

axm9 : $d5 \neq d6 \wedge d5 \neq d7 \wedge d5 \neq d8 \wedge d5 \neq d9 \wedge d5 \neq d10$

axm10 : $d6 \neq d7 \wedge d6 \neq d8 \wedge d6 \neq d9 \wedge d6 \neq d10$

axm11 : $d7 \neq d8 \wedge d7 \neq d9 \wedge d7 \neq d10$

axm12 : $d8 \neq d9 \wedge d8 \neq d10$

axm13 : $d9 \neq d10$

END

A.1.2 Máquina

An Event-B Specification of maquina1
Generated Date: 26 Jan 2010 @ 05:40:53 PM

MACHINE maquina1

SEES contexto1

VARIABLES

canalto

canalfrom

buf

contadorinserbuffer

contadorremovbuf

dadosrecebidos

block

blockrecv

INVARIANTS

inv1 : $canalto \in dados$

inv2 : $canalfrom \in dados$

inv3 : $buf \in 0 \mathinner{..} tamanho \rightarrow dados$

inv4 : $contadorinserbuffer \leq tamanho$

inv5 : $contadorinserbuffer \in \mathbb{N}$

inv6 : $contadorremovbuf \in \mathbb{N}$

inv7 : $contadorremovbuf \leq tamanho$

inv8 : $dadosrecebidos \in dados$

inv9 : $block \in \{0, 1\}$

inv10 : $blockrecv \in \{0, 1\}$

EVENTS

Initialisation

begin

act1 : $canalto := vazio$

act2 : $canalfrom := vazio$

act4 : $buf := 0 \mathinner{..} tamanho \times \{vazio\}$

act5 : $contadorinserbuffer := 0$

act6 : $contadorremovbuf := 0$

act7 : $dadosrecebidos := vazio$

act8 : $block := 0$

act9 : $blockrecv := 0$

end

Event *send* $\widehat{=}$

any

$mensagem$

where

grd1 : $mensagem \in dados$

grd2 : $canalto = vazio$

grd3 : $mensagem \neq vazio$

grd4 : $block = 0$

then

act2 : $canalto := mensagem$

end

Event *bufrecebe* $\widehat{=}$

when

grd1 : $canalto \neq vazio$

grd2 : $contadorinserbuffer < tamanho$

grd3 : $buf(contadorinserbuffer) = vazio$

then

act1 : $buf(contadorinserbuffer) := canalto$

act2 : $contadorinserbuffer := contadorinserbuffer + 1$

act3 : $canalto := vazio$

end

Event *bufrecebe2* $\widehat{=}$

when

grd1 : $canalto \neq vazio$

grd2 : $contadorinserbuffer = tamanho$

grd3 : $vazio \in ran(buf)$

grd4 : $buf(contadorinserbuffer) = vazio$

then

act1 : $buf(contadorinserbuffer) := canalto$

act2 : $canalto := vazio$

act3 : $contadorinserbuffer := 0$

end

Event *bufretira* $\widehat{=}$

when

grd1 : $canalfrom = vazio$

grd2 : $contadorremovbuf < tamanho$

grd3 : $buf(contadorremovbuf) \neq vazio$

then

act1 : $canalfrom := buf(contadorremovbuf)$

act2 : $contadorremovbuf := contadorremovbuf + 1$

act3 : $buf(contadorremovbuf) := vazio$

end

Event *recv* $\widehat{=}$

when

grd1 : $blockrecv = 0 \lor (blockrecv = 1 \land canalfrom \neq vazio)$

then

act1 : $dadosrecebidos := canalfrom$

act2 : $canalfrom := vazio$

act3 : $blockrecv := 0$

end

Event *bufretira2* $\widehat{=}$

when

grd1 : $canalfrom = vazio$

grd2 : $contadorremovbuf = tamanho$

grd3 : $buf(contadorremovbuf) \neq vazio$

then

act1 : $canalfrom := buf(contadorremovbuf)$

act2 : $contadorremovbuf := 0$

act3 : $buf(contadorremovbuf) := vazio$

end

Event *sendblock* $\widehat{=}$

any

$mensagem$

where

grd1 : $mensagem \in dados$

grd2 : $canalto = vazio$

grd3 : $mensagem \neq vazio$

grd4 : $block = 0$

then

act1 : $canalto := mensagem$

act2 : $block := 1$

end

Event *desblocksend* $\widehat{=}$

when

grd1 : $canalto = vazio$

grd2 : $buf[0\,..\,tamanho] = \{vazio\}$

grd3 : $block = 1$

grd4 : $canalfrom = vazio$

then

act1 : $block := 0$

end

Event *recvblock* $\widehat{=}$

when

grd1 : $blockrecv = 0$

then

act1 : $blockrecv := 1$

end

END

A.2 Exclusão Mútua

Nesta seção são apresentados o contexto (Seção A.1.1) e a máquina (Seção A.1.2) do algoritmo de exclusão mútua centralizado.

Em [TS07] é proposto um algoritmo de exclusão mútua centralizado. Neste algoritmo existe um coordenador que recebe os pedidos de recursos dos processo e decide quais processos devem usar o recurso. Para tanto sempre que um processo necessita acessar um recurso, envia uma mensagem de requisição para o coordenador indicando que recurso precisa. Se nenhum processo estiver utilizando o recurso, o coordenador envia uma mensagem para o processo concedendo o recurso. Entretanto se outro processo estiver usando o recurso, o coordenador registra o pedido do processo em uma fila e não envia nada como resposta, deixando o processo esperando.

Através de uma fila de requisições é possivel garantir que os recursos serão liberados para os processos por ordem de chegada, ou seja, o processo que requisitar primeiro determinado recurso estará antes na fila, portanto será atendido primeiro. Quando um processo conclui a utilização do

recurso, envia uma mensagem ao coordenador avisando-o. O coordenador examina a fila de requisições para verificar se existe algum pedido para este recurso. Existindo um pedido, o coordenador envia uma mensagem para o processo que fez a requisição do recurso, avisando ao mesmo que pode utilizar o recurso liberado. Caso não existe uma requisição para este recurso na fila, o coordenador deixa o recurso disponível para uma futura requisição por parte dos processos.

Além do evento de inicialização, o algoritmo exclusão mútua contém os seguintes eventos modelados: *pedidoderecurso*, *coordrecurso*, *alocarecurso*, *processoativarecurso*, *coordregistrapedido*, *procliberarecurso*, *coordliberarecurso*, *coordverificarecurso1*, *coordenviarecurso*, *coordavancavetor*.

O evento *pedidoderecurso* é responsável pelo envio dos pedidos de recursos dos processo para o líder. O evento *coordrecurso* é responsável por receber os pedidos de recurso no líder. O evento *alocarecurso* é responsável por alocar e enviar os recursos para os processos. O evento *processoativarecurso* representa a recepção dos recursos, pelos processos participantes. Para recursos ocupados, o evento *coordregistrapedido* registra os pedidos em uma fila. O evento *procliberarecurso* representa a liberação dos recursos, pelos processos. O evento *coordliberarecurso* é responsável por receber o aviso de liberação de recursos dos processos. O evento *coordverificarecurso1* é responsável por verificar, em determinado endereço da fila de pedidos, se o recurso liberado não foi pedido por outro processo. O evento *coordenviarecurso* é responsável por enviar um recurso liberado, para um processo encontrado na fila de espera. O evento *coordavancavetor* é responsável por avançar em uma unidade o endereço de verificação da fila no evento *coordverificarecurso*.

A.2.1 Contexto

An Event-B Specification of contexto1
Generated Date: 26 Jan 2010 @ 05:38:34 PM

CONTEXT contexto1

SETS

estadosdosprocessos

estadosdosrecursos

mensagens

CONSTANTS

comrecurso

semrecurso

ocupado

livre

np

aguardandorecurso

nr

permissao

liberacao

tamanhovetor

vazio

AXIOMS

axm1 : $estadosdosprocessos = \{comrecurso, aguardandorecurso, semrecurso\}$

axm2 : $estadosdosrecursos = \{ocupado, livre\}$

axm3 : $comrecurso \neq semrecurso$

axm4 : $ocupado \neq livre$

axm5 : $np = 5$

axm6 : $comrecurso \neq aguardandorecurso$

axm7 : $aguardandorecurso \neq semrecurso$

axm8 : $nr = 2$

axm9 : $mensagens = \{permissao, liberacao, vazio\}$

axm10 : $permissao \neq liberacao$

axm11 : $tamanhovetor = 10$

axm12 : $permissao \neq vazio$

axm13 : $liberacao \neq vazio$

END

A.2.2 Máquina

An Event-B Specification of refinamento1
Generated Date: 26 Jan 2010 @ 05:38:37 PM

MACHINE refinamento1

SEES contexto1

VARIABLES

canalid

canalrecurso

fasesdosprocessos

bufrecurso

bufid

recursos

vetoresperarecursos

vetoresperaid

contadorinserir

contadorretirada

canallider

canallider2

recursoativo

canalprocmensagem

INVARIANTS

inv1 : $canalid \in 0 \mathinner{..} np$

inv2 : $canalrecurso \in 0 \mathinner{..} nr$

inv3 : $fasesdosprocessos \in 1 \mathinner{..} np \rightarrow estadosdosprocessos$

inv5 : $bufrecurso \in 0 \mathinner{..} nr$

inv6 : $bufid \in 0 \mathinner{..} np$

inv7 : $recursos \in 1 \mathinner{..} nr \rightarrow estadosdosrecursos$

inv8 : $vetoresperarecursos \in 0 \mathinner{..} tamanhovetor \rightarrow 0 \mathinner{..} nr$

inv9 : $vetoresperaid \in 0 \mathinner{..} tamanhovetor \rightarrow 0 \mathinner{..} np$

inv10 : $contadorinserir \in 0 \mathinner{..} tamanhovetor$

inv11 : $contadorretirada \in 0 \mathinner{..} tamanhovetor$

inv12 : $canallider \in 0 \mathinner{..} np$

inv13 : $canallider2 \in 0 \mathinner{..} nr$

inv14 : $recursoativo \in 1 \mathinner{..} np \rightarrow 0 \mathinner{..} nr$

inv15 : $canalprocmensagem \in mensagens$

EVENTS

Initialisation

begin

act2 : $canalrecurso := 0$

act1 : $canalid := 0$

act3 : $fasesdosprocessos := 1 \mathinner{..} np \times \{semrecurso\}$

act5 : $bufrecurso := 0$

act6 : $bufid := 0$

act7 : $recursos := 1 \mathinner{..} nr \times \{livre\}$

act8 : $vetoresperarecursos := 0 \mathinner{..} tamanhovetor \times \{0\}$

act9 : $vetoresperaid := 0 \mathinner{..} tamanhovetor \times \{0\}$

act10 : $contadorinserir := 0$

act11 : $contadorretirada := 0$

act12 : $canallider := 0$

act13 : $canallider2 := 0$

act14 : $recursoativo := 1 \mathinner{..} np \times \{0\}$

act15 : $canalprocmensagem := vazio$

end

Event *pedidoderecurso* $\widehat{=}$

any

$processo$

$recurso$

where

grd1 : $processo \in 1 \mathinner{..} np$

grd2 : $recurso \in 1 \mathinner{..} nr$

grd3 : $canalid = 0$

grd4 : $canalrecurso = 0$

grd5 : $fasesdosprocessos(processo) = semrecurso$

grd6 : $canallider = 0$

grd7 : $canallider2 = 0$

grd8 : $canalprocmensagem = vazio$

then

act1 : $canalid := processo$

act2 : $canalrecurso := recurso$

act3 : $fasesdosprocessos(processo) := aguardandorecurso$

act4 : $canalprocmensagem := permissao$

end

Event *coordrecursos* $\widehat{=}$

when

grd1 : $canalid \neq 0$

grd2 : $canalrecurso \neq 0$

grd3 : $bufid = 0$

grd4 : $bufrecurso = 0$

grd5 : $canalprocmensagem = permissao$

then

act1 : $bufrecurso := canalrecurso$

act2 : $bufid := canalid$

end

Event *alocarecursos* $\widehat{=}$

when

grd1 : $canalid \neq 0$

grd2 : $canalrecurso \neq 0$

grd3 : $bufid \neq 0$

grd4 : $bufrecurso \neq 0$

grd5 : $recursos(bufrecurso) = livre$

grd6 : $canalprocmensagem = permissao$

then

act1 : $recursos(bufrecurso) := ocupado$

act2 : $canallider := bufid$

act3 : $canallider2 := bufrecurso$

act4 : $canalrecurso := 0$

act5 : $canalid := 0$

act6 : $bufrecurso := 0$

act7 : $bufid := 0$

act8 : $canalprocmensagem := vazio$

end

Event *processoativarecurso* $\widehat{=}$

when

grd1 : $canalid = 0$

grd2 : $canalrecurso = 0$

grd3 : $canallider \neq 0$

grd4 : $canallider2 \neq 0$

grd5 : $fasesdosprocessos(canallider) = aguardandorecurso$

grd6 : $canalprocmensagem = vazio$

then

act1 : $fasesdosprocessos(canallider) := comrecurso$

act2 : $recursoativo(canallider) := canallider2$

act3 : $canallider := 0$

act4 : $canallider2 := 0$

end

Event *coordregistrapedido* $\widehat{=}$

when

grd1 : $canalid \neq 0$

grd2 : $canalrecurso \neq 0$

grd3 : $bufid \neq 0$

grd4 : $bufrecurso \neq 0$

grd5 : $recursos(bufrecurso) = ocupado$

grd6 : $canalprocmensagem = permissao$

grd7 : $contadorinserir < tamanhovetor$

grd8 : $vetoresperaid(contadorinserir) = 0$

then

act2 : $vetoresperaid(contadorinserir) := bufid$

act3 : $vetoresperarecursos(contadorinserir) := bufrecurso$

act1 : $contadorinserir := contadorinserir + 1$

act4 : $canalrecurso := 0$

act5 : $canalid := 0$

act6 : $bufrecurso := 0$

act7 : $bufid := 0$

act8 : $canalprocmensagem := vazio$

end

Event *procliberarecurso* $\widehat{=}$

any

$processo$

$recurso$

where

grd1 : $processo \in 1 \mathinner{..} np$

grd2 : $fasesdosprocessos(processo) = comrecurso$

grd3 : $canalid = 0$

grd4 : $canalrecurso = 0$

grd5 : $canalprocmensagem = vazio$

grd6 : $canallider = 0$

grd7 : $canallider2 = 0$

grd9 : $recurso \in 1 \mathinner{..} nr$

grd8 : $recurso = recursoativo(processo)$

then

act1 : $fasesdosprocessos(processo) := semrecurso$

act2 : $canalid := processo$

act3 : $canalrecurso := recurso$

act4 : $canalprocmensagem := liberacao$

act5 : $recursoativo(processo) := 0$

end

Event *coordliberarecurso* $\widehat{=}$

when

grd1 : $canalid \neq 0$

grd2 : $canalrecurso \neq 0$

grd3 : $bufid = 0$

grd4 : $bufrecurso = 0$

grd5 : $canalprocmensagem = liberacao$

then

act1 : $bufid := canalid$

act2 : $bufrecurso := canalrecurso$

act3 : $recursos(canalrecurso) := livre$

end

Event *coordverificarecurso1* $\widehat{=}$

when

grd1 : $canalid \neq 0$

grd2 : $canalrecurso \neq 0$

grd3 : $bufid \neq 0$

grd4 : $bufrecurso \neq 0$

grd5 : $canalprocmensagem = liberacao$

grd6 : $bufrecurso \notin ran(vetoresperarecursos)$

then

act1 : $canalid := 0$

act2 : $canalrecurso := 0$

act3 : $bufid := 0$

act4 : $bufrecurso := 0$

act5 : $canalprocmensagem := vazio$

end

Event *coordenviarecurso* $\widehat{=}$

when

grd1 : $canalid \neq 0$

grd2 : $canalrecurso \neq 0$

grd3 : $bufid \neq 0$

grd4 : $bufrecurso \neq 0$

grd5 : $canalprocmensagem = liberacao$

grd6 : $bufrecurso \in ran(vetoresperarecursos)$

grd7 : $vetoresperarecursos(contadorretirada) = bufrecurso$

then

act1 : $recursos(bufrecurso) := ocupado$

act2 : $canallider := vetoresperaid(contadorretirada)$

act3 : $canallider2 := vetoresperarecursos(contadorretirada)$

act4 : $canalrecurso := 0$

act5 : $canalid := 0$

act6 : $bufrecurso := 0$

act7 : $bufid := 0$

act8 : $canalprocmensagem := vazio$

act9 : $vetoresperarecursos(contadorretirada) := 0$

act10 : $vetoresperaid(contadorretirada) := 0$

act11 : $contadorretirada := 0$

end

Event *coordavancavetor* $\widehat{=}$

when

grd1 : $canalid \neq 0$

grd2 : $canalrecurso \neq 0$

grd3 : $bufid \neq 0$

grd4 : $bufrecurso \neq 0$

grd5 : $canalprocmensagem = liberacao$

grd6 : $bufrecurso \in ran(vetoresperarecursos)$

grd7 : $vetoresperarecursos(contadorretirada) \neq bufrecurso$

grd8 : $contadorretirada < tamanhovetor$

then

act1 : $contadorretirada := contadorretirada + 1$

end

END

A.3 Comunicação em Grupo

Nesta seção são apresentados o contexto (Seção A.1.1) e a máquina (Seção A.1.2) do algoritmo de comunicação em grupo modelado em Event-B.

Para este algoritmo, um dado processo *p* pode enviar uma mensagem de diferentes modos: envio *broadcast*, *multicast* ou *unicast*.

Dado *P* como o processo responsável pelo envio de uma determinada mensagem *M* e *G* como o grupo de processos ativos, mensagens em *broadcast* são enviadas de modo *G-P*, ou seja, todos os processos ativos recebem a mensagem exceto o processo P que a enviou.

Para o envio da mensagem em modo *multicast* são definidos grupos de processos, portanto cada processo podendo ou não estar em um ou mais grupos. Dado *GA* e *GB* como grupos de processos, *P* como o processo que envia uma determinada mensagem *M*, mensagens em *multicast* são enviadas de modo *GA-P* (*multicast* para grupo *GA*) ou *GB-P* (*multicast* para grupo *GB*), ou seja, se o *multicast* for para o grupo *GA* todos os processos participantes do grupo *GA* recebem a mensagem *M*, exceto o processo *P* que a enviou. Entretanto se for realizado um *multicast* para grupo *GB* todos os processos do grupo *GB* recebem a mensagem *M*, exceto o processo P que a enviou.

O envio de uma mensagem ponto-a-ponto pode ser chamado de *unicast*, ou seja, uma mensagem *M* é enviada de um processo *P* para um processo *R*.

Além do evento de inicialização, o algoritmo de comunicação em grupo, contém os seguintes eventos: *inserirgrupoA*, *inserirgrupoB*, *removergrupoA*, *removergrupoB*, *enviamulticastA*, *enviamulticastB*, *recbmulticast*, *broadcast*, *unicast*.

Os eventos *inserirgrupoA* e *inserirgrupoB* são responsáveis por adicionar processos, respectivamente, nos grupos *multicast grupoA* e *grupoB*. Os eventos *removergrupoA* e *removergrupoB* são

responsáveis por remover processos, respectivamente, nos grupos *multicast grupoA* e *grupoB*. Os eventos *enviamulticastA* e *enviamulticastB* são responsáveis por enviar mensagens do tipo *multicast*, respectivamente, para os grupos *grupoA* e *grupoB*. O evento *broadcast* é responsável por enviar mensagens do tipo *broadcast* para os processos e o evento *unicast*, por enviar mensagens do tipo *unicast* para os processos. Já o evento *recbmulticast* é responsável pela recepção, pelos processos, de todos os tipos de mensagens.

A.3.1 Contexto

An Event-B Specification of contexto1
Generated Date: 26 Jan 2010 @ 05:26:17 PM

CONTEXT contexto1

SETS

dados

CONSTANTS

np

d1

d2

vazio

d3

AXIOMS

axm1 : $np = 3$

axm3 : $dados = \{vazio, d1, d2, d3\}$

axm4 : $d1 \neq d2 \land d1 \neq d3$

axm5 : $vazio \neq d1 \land vazio \neq d2 \land vazio \neq d3$

axm6 : $d2 \neq d3$

END

A.3.2 Máquina

An Event-B Specification of maquina1
Generated Date: 26 Jan 2010 @ 05:26:15 PM

MACHINE maquina1

SEES contexto1

VARIABLES

grupoA

grupoB

mensagensrecebidas

canalmensagem

canalgrupo

INVARIANTS

inv1 : $grupoA \subseteq 1 \mathinner{..} np$

inv2 : $grupoB \subseteq 1 \mathinner{..} np$

inv3 : $mensagensrecebidas \in 1 \mathinner{..} np \rightarrow dados$

inv4 : $canalmensagem \in dados$

inv5 : $canalgrupo \subseteq 1 \mathinner{..} np$

EVENTS

Initialisation

begin

act1 : $grupoA := \varnothing$

act2 : $grupoB := \varnothing$

act3 : $mensagensrecebidas := 1 \mathinner{..} np \times \{vazio\}$

act4 : $canalmensagem := vazio$

act5 : $canalgrupo := \varnothing$

end

Event *inserirgrupoA* $\widehat{=}$

any

$processo$

where

grd1 : $processo \in 1 \mathinner{..} np$

grd2 : $processo \notin grupoA$

then

act1 : $grupoA := grupoA \cup \{processo\}$

end

Event *inserirgrupoB* $\widehat{=}$

any

$processo$

where

grd1 : $processo \in 1 \mathinner{..} np$

grd2 : $processo \notin grupoB$

then

act1 : $grupoB := grupoB \cup \{processo\}$

end

Event *removergrupoA* $\widehat{=}$

any

$processo$

where

grd1 : $processo \in 1 \mathinner{..} np$

grd2 : $processo \in grupoA$

then

act1 : $grupoA := grupoA \setminus \{processo\}$

end

Event *removergrupoB* $\widehat{=}$

any

$processo$

where

grd1 : $processo \in 1 \mathinner{..} np$

grd2 : $processo \in grupoB$

then

act1 : $grupoB := grupoB \setminus \{processo\}$

end

Event *enviamulticastA* $\widehat{=}$

any

$processo$

$mensagem$

where

grd1 : $processo \in 1 \mathinner{..} np$

grd2 : $mensagem \in dados \land mensagem \neq vazio$

grd3 : $grupoA \neq \varnothing$

then

act1 : $canalmensagem := mensagem$

act2 : $canalgrupo := grupoA \setminus \{processo\}$

end

Event *enviamulticastB* $\widehat{=}$

any

$processo$

$mensagem$

where

grd1 : $processo \in 1 \mathinner{..} np$

grd2 : $mensagem \in dados \land mensagem \neq vazio$

grd3 : $grupoB \neq \varnothing$

then

act1 : $canalmensagem := mensagem$

act2 : $canalgrupo := grupoB \setminus \{processo\}$

end

Event *recbmulticast* $\widehat{=}$

any

$processo$

where

grd1 : $canalmensagem \neq vazio$

grd2 : $canalgrupo \neq \varnothing$

grd3 : $processo \in canalgrupo$

then

act1 : $mensagensrecebidas(processo) := canalmensagem$

act2 : $canalgrupo := canalgrupo \setminus \{processo\}$

end

Event *broadcast* $\widehat{=}$

any

$processo$

$mensagem$

where

grd1 : $processo \in 1 \,..\, np$

grd2 : $mensagem \in dados$

grd3 : $mensagem \neq vazio$

then

act1 : $canalmensagem := mensagem$

act2 : $canalgrupo := 1 \,..\, np$

end

Event *unicast* $\widehat{=}$

any

$processosend$

$processorecv$

$mensagem$

where

grd1 : $processosend \in 1 \,..\, np$

grd2 : $processorecv \in 1 \,..\, np$

grd3 : $mensagem \in dados \land mensagem \neq vazio$

grd4 : $processosend \neq processorecv$

then

act1 : $canalmensagem := mensagem$

act2 : $canalgrupo := \{processorecv\}$

end

END