Sistemas Tolerantes a Falhas

Ténicas de TF para Projeto de Diversidade

Prof. Jó Ueyama

Introdução

- Diversidade é implementada por várias técnicas
- A técnica predominante
  - Variantes
  - Evita erros coincidentes
  - Se ocorre erros mesmo assim?
    - Facilita na detecção e na resolução de erros
- Técnica idealizada por pessoas históricas como Charles Babbage
- Duas técnicas básicas a serem discutidas
  - Recovery block (RcB)
  - N-Version Programming

Recovery Blocks

- É uma das primeiras técnicas da diversidade
- É uma técnica considerada como dinâmica
  - A seleção do *output* da variante pelo 'juiz' é realizada durante a execução
    - Baseado no AT (acceptance test)
- RcB usa o AT e o *backward recovery* para efetuar a recuperação
- A execução dos sistemas podem ser realizadas através de diversos algoritmos e projetos
- Várias funções possuem critérios diferentes em termos de eficiência, segurança e tempo

Recovery Block

- O RcB permite o uso de critérios diferentes que são selecionados dinamicamente
- Primary block
  - Variante mais eficiente executado primeiro
- Os menos eficientes:
  - Dispostos serialmente
  - Conhecidos como secundários ou alternativas
- As variantes são dispostos na ordem de 'degradações' em termos de performance
  - Ou segurança, por exemplo.

Funcionamento do RcB

- RcB possui:
  - um executor (coordenador);
  - um *acceptance test;*
  - um bloco primário e;
  - blocos de tentivas (*retry blocks*)
- Muitas implementações possuem também
  - *Watchdog* timer (WDT)
  - Particularmente em sistemas de tempo real

Funcionamento de RcB

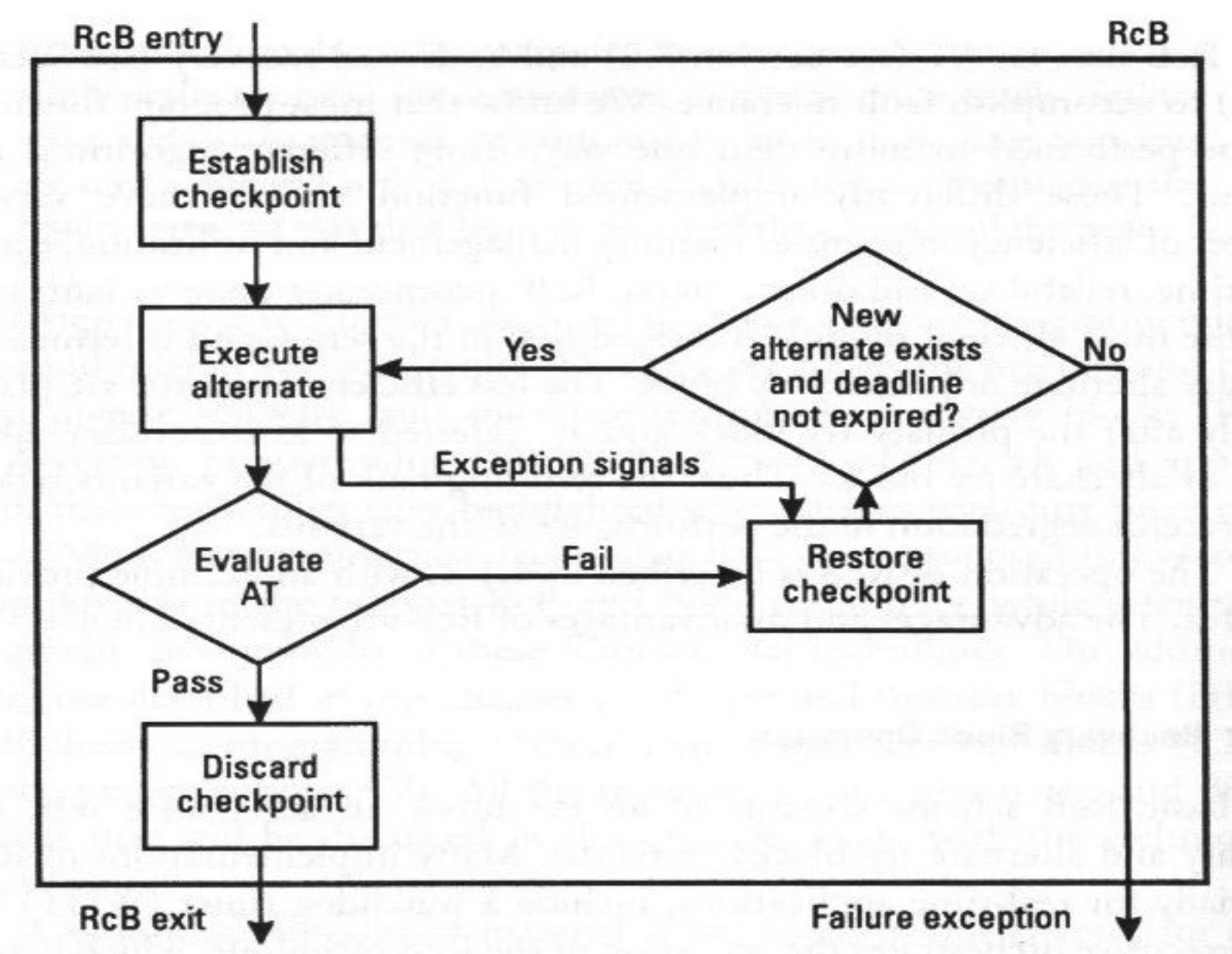
- O código RcB abaixo:
  - Assegura que o output do bloco primário passa no teste do AT
  - Caso contrário, todas as alternativas (variantes) serão executadas e o seu output testado pelo AT
  - Caso nenhuma passe no AT, então levanta a falha

```
ensure          Acceptance Test
by              Primary Alternate
else by         Alternate 2
else by         Alternate 3
...
else by         Alternate n
else failure exception
```

Funcionamento e estrutura do RcB

- A próxima figura ilustra a estrutura e o funcionamento do RcB com WDT
- Vários cenários são apresentados
  - *Failure-free operation*
  - *Exception or timeout in primary alternate execution*
  - *Primary's results on time, but fails AT; successful alternate execution*
  - *All alternates exhausted without success*

Funcionamento e estrutura do RcB



RcB Estendido

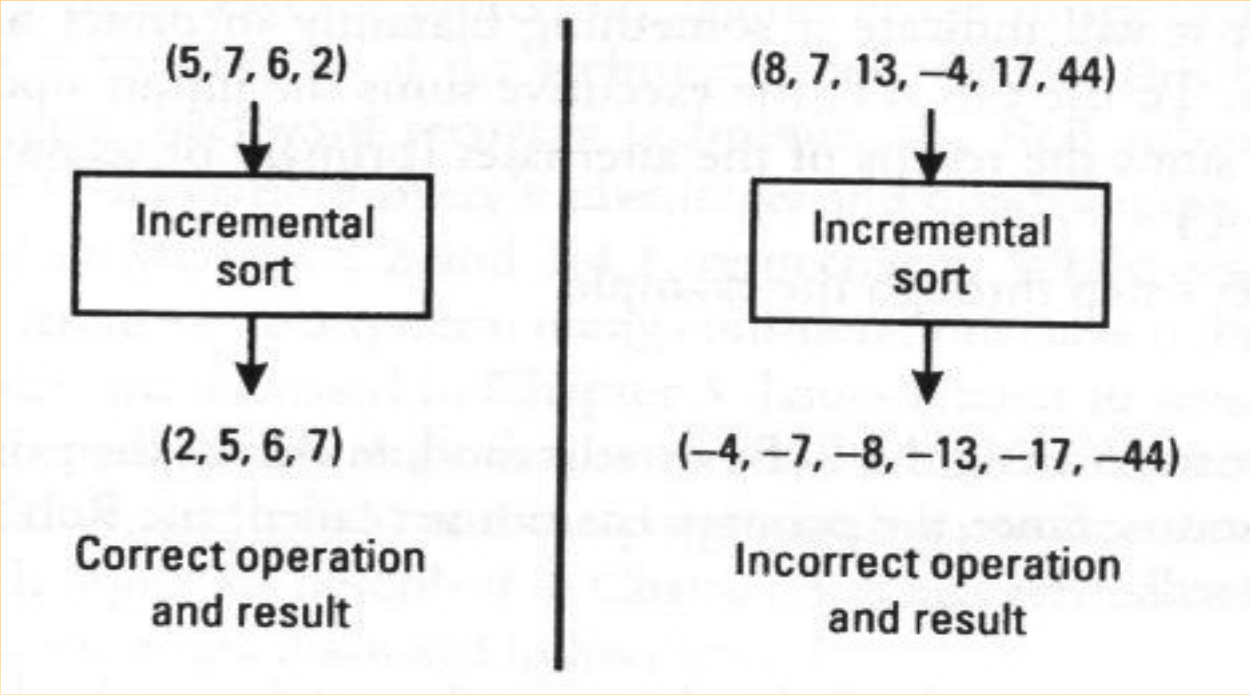
- Existem vários mecanismos estendendo o RcB original
- Uso do WDT
- *Alternate execution counter*
  - Indica o # de vezes que as variantes devem ser executadas (a própria e/ou as próximas variantes)
  - Permite retirar o bloco primário temporariamente para: substituir ou fazer 'reparos'

RcB Estendido

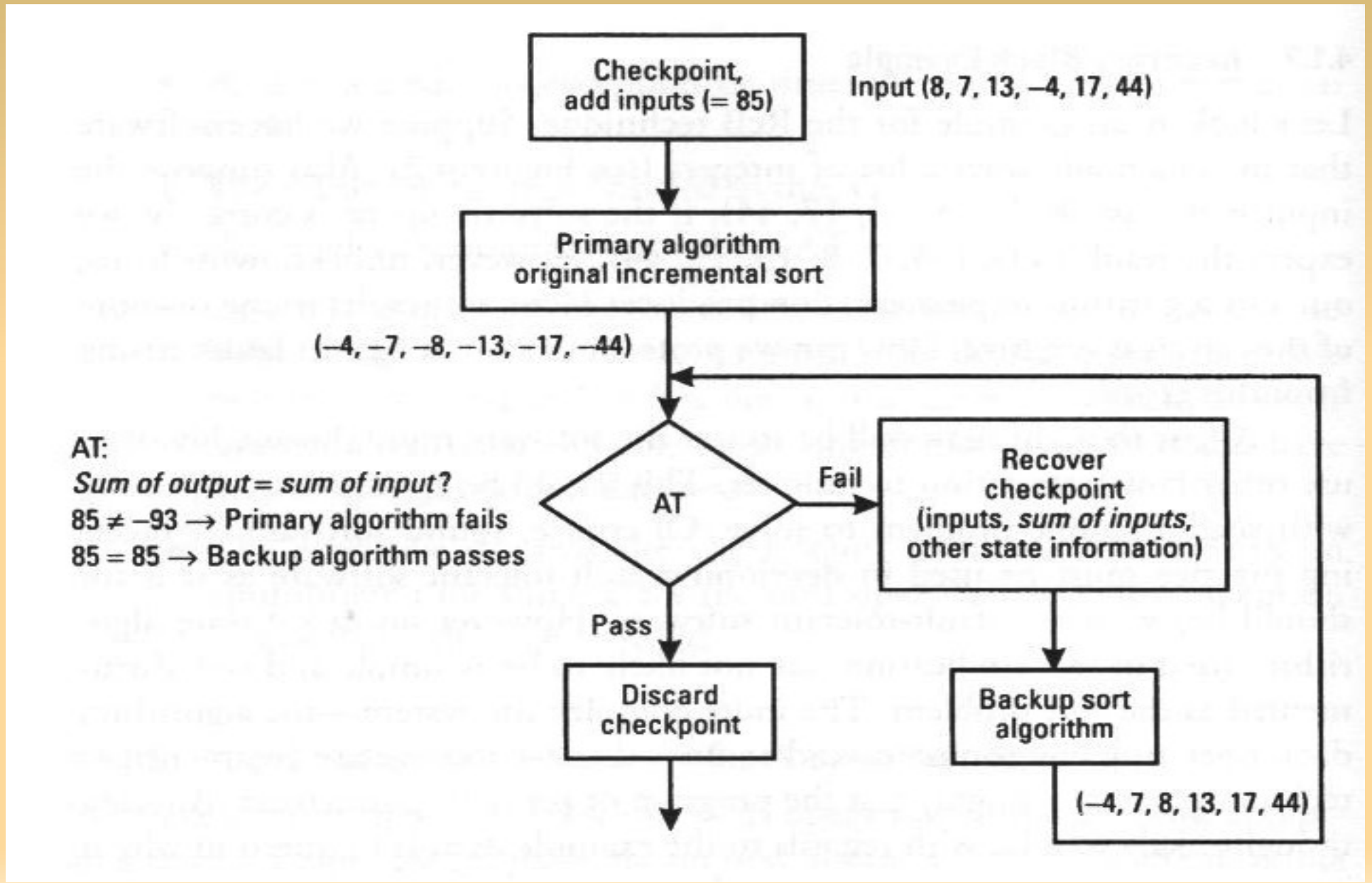
- Provisão de ATs mais sofisticados que realizam diversos testes
- Um AT é chamado antes só para validar os parâmetros recebidos (e.g. formato)
  - Caso falhe, já invoca a próxima variante
  - Ou nem invoca as variantes, pois os dados estão com falhas
  - Neste caso, pode fazer o uso de dados de default ou de backup
  - Ou RcB pode implementar uma variante só para derivar dados de entrada

Exemplo de RcB

- Exemplo: uma aplicação para fazer a ordenação de inteiros
- Lista de números: 8, 7, 13, -4, 17, 44
- Output correto: -4, 7, 8, 13, 17, 44
- Suponha que ele produz erros para n < 0



Exemplo de Implementação do RcB



Algumas Considerações do Exemplo

- O AT não resolve problemas de erros de ordenação; o AT é bem simples
- O Controlador dirige a execução do RcB:
  - Guardando os dados do Checkpoint
  - Descartando-os
  - Executando o AT e validando os outpus gerados
  - Invocando as variantes
    - quando for o caso

Discussão do RcB

- Como outros sistemas de TF, o RcB protege a tradução dos requisitos para o código
- Mas, o RcB não protege a especificação de requisitos
- O RcB é executado em um ambiente uniprocessor de maneira sequencial
- O ovehead desta técnia consiste em:
  - Armazenar os dados do Checkpoint
  - Executar o AT
  - Executar as variantes caso o primary block falhe

Discussão do RcB

- Na maior parte das vezes, o primary executa corretamente, o overhead é:
  - Executar o AT
  - *Manage* Checkpoint
- Variação de tempo de execução não é benéfica para sistemas de tempo real
  - Pode só o primary block como todas as variantes (dependendo do output)
- Uma solução para isso é o DRB (Distributed Recovery Block)
- Variantes e ATs podem ser executados em paralelo

Discussão do RcB

- Vantagem do RcB
  - Pode suportar um *fine grained* TF, onde partes dos módulos críticos podem ser 'testados'
  - Baixo custo
- O ponto chave deste mecanismo é o AT
- Potencial problema do método: overhead com o Checkpoint
  - Efeito dominó (efeito cascata do rollback)
  - Falta de coordenação entre os rollbacks
  - *Conversation Scheme* como forma de evitar o efeito dominó

Discussão do RcB

- Para implementar o RcB várias técnicas podem ser usadas:
  - Assertions
  - Checkpointing
  - Atomic actions
  - Idealized components
- Depende da arquitetura e da aplicação (e.g. *performance issues*)

Técnica da Conversação

- A técnica da conversação é um mecanimo para evitar o efeito dominó
- É um mecanismo que permite o processamento concorrente de vários grupos de processos
  - Podem ser executados de forma segura
  - Os grupos são chamados de *conversation*
- Cada *conversation* só podem comunicar-se dentro do grupo
  - Não é permitido trocar informações entre os grupos
  - Para entrar no grupo cada processo necessita ter sido '*checkpointed*'

Técnica da Conversação

- Cada *conversation* possui um AT que é executado após o término de todos os processos
- Se falha (AT), então todos os processos fazem o rollback e executam as suas próprias variantes
- Se sucede, então descartam os checkpoints e saem de forma síncrona da *conversation*
- *Conversations* são unidades de rollback dos processos
- O AT do *conversation* consiste em uma avaliação definida para o grupo como um todo

Técnicas de Conversação

- Limitações da 'Conversação'
- O desenvolvedor deve criar os grupos e os pontos de recuperação (*recovery points*)
- O programador deve criar AT globais para cada *conversation*
  - Tal AT é uma combinação dos requisitos de cada processo
- Esta técnica envolve um overhead para que grupos e processos se sincronizem
  - e.g. para que os processos encontrem-se para um AT
- Processos podem sair do *conversation*
  - e.g. deadline

Concluindo RcB

- Existem várias versões estendidas que incluem prover transparências
  - e.g. coordenação transparente para os programadores

Sistemas Tolerantes a Falhas

Programação N-version

Prof. Jó Ueyama

N-Version Programming

- RcB e NVP são consideradas as técnicas clássicas da diversidade
- A técnica é antiga e surgiu em 1977
- É considerada uma técnica estática. Por que?
  - Uma tarefa é executada por *n* processos cujos resultados serão analisados por um juiz
  - Voto da maioria (normalmente)
  - O fluxo da execução não altera com o resultado
- Normalmente as versões são executadas concorrentemente

NVP

- NVP possui um módulo de decisão (DM) e usa a técnica do *forward recovery* para prover a recuperação
- Usa a abordagem das variantes
- DM seleciona a 'melhor' saída
- A técnica é composta de:
  - Executor (ou coordenador)
  - *n* variantes
  - DM (*decision mechanism*)

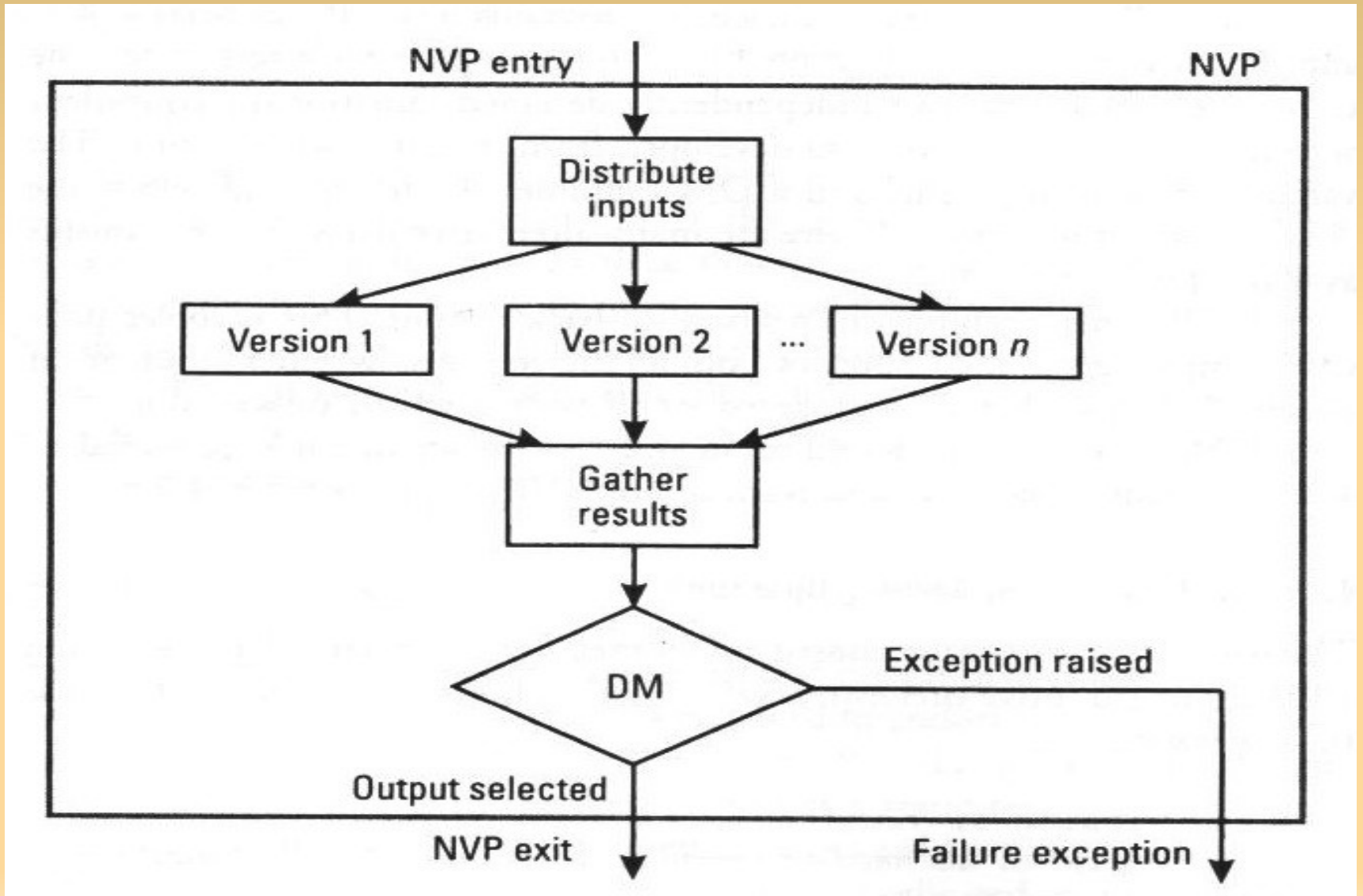
NVP

- O coordenador executa a operação da NVP com a seguinte sintaxe
  - As versões são executadas em paralelo
  - Resultados são recebidos das versões que são por sua vez enviados como parâmetros de um DM
  - DM verifica se uma resposta pode ser escolhida
  - Caso contrário uma exceção é levantada

```
run Version 1, Version 2, ..., Version n
if (Decision Mechanism (Result 1, Result 2, ..., Result n))
    return Result
else failure exception
```

NVP

- Estrutura e operação do NVP



NVP Failure free operation

- Nenhum erro ocorre na execução das variantes
- O executor envia os dados de entrada às variantes
- Variantes disponibilizam resultados iguais
- DM pega randomicamente a resposta da variante 2, por exemplo
- Controle volta ao executor
  - Retorna o resultado correto ao módulo fora do NVP

Failure scenario – incorrect results

- Inicialmente os dados de entrada passam pelo teste do formato
- Resultados são recebidos de todas as variantes
- O executor coleta os resultados e os submete ao DM
  - DM verifica que os resultados diferem muito um do outro
  - Voto da maioria não funciona, quando os resultados são muito diferentes
- Resultado: uma exceção é levantada

Failure scenario – version does not execute

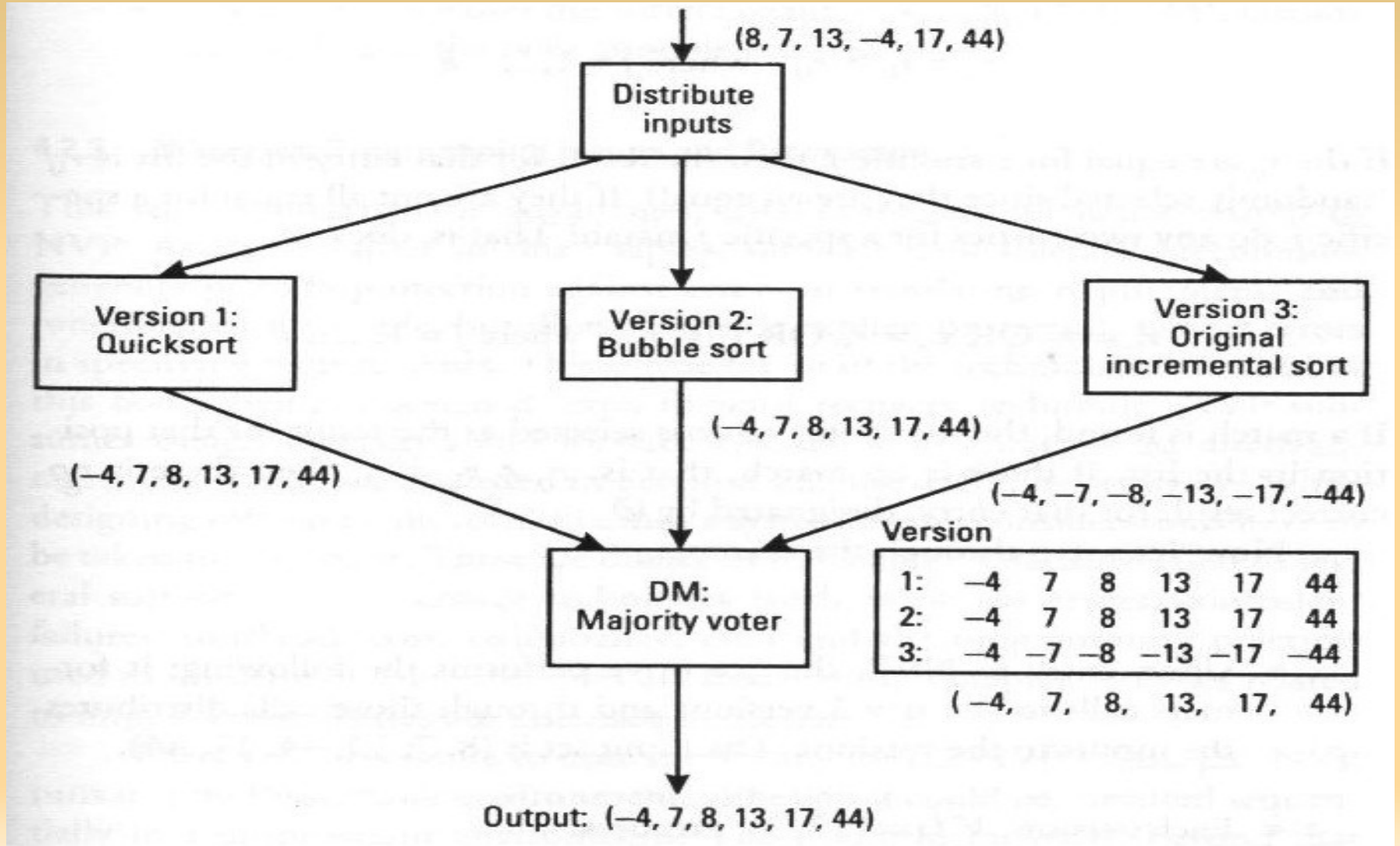
- Após o teste do formato o executor envia os dados a cada uma das *n* variantes
- Uma ou mais variantes deixam de fornecer resultados (e.g. *endless loop*)
- Se a DM não puder manusear menos do que *n* resultados?
  - Deixa um flag que foi incapaz de escolher um resulatdo correto baseado no voto da maioria
- Exceção é levantada

Extensões/Melhoras no NVP

- A maior parte das extensões dizem respeito aos mecanismo de decisão (DM)
- *Dynamic voter*: capaz de manusear uma quantidade variável de resultados
  - Eficiente no problema anterior
- Um outro mecanismo é não esperar o fim de todas as variantes
  - Se chegam dois resultados, vota pela maioria; caso haja resultados diferentes, espera pelo terceiro
  - Mais rápido do que o método básico

Exemplo de Programação NVP

- Exemplo é a mesma aplicação do sort que produz erros qd os elementos < 0 (negativos)



Discussão sobre o NVP

- NVP carrega as vantagens e desvantages do *forward recovery*
  - Não necessita de um tempo extra para fazer o rollback (vantagem)
  - É bem específico para cada aplicação (desvantagem)
- NVP é voltado para ambientes multiprocessados
  - Mas pode ser executado em ambientes sequenciais
- Overhead para executar as *n* variantes, o executor e o DM
- Overhead para realizar a sincronização. Por que?

Discussão sobre o NVP

- O gargalo desta técnica está em esperar pela variante mais 'lenta'
- Usar métodos de reconfiguração dinâmica
- Não apenas os desenvolvedores devem ser distintos, mas os que fazem a manutenção
  - Das variantes
- NVP não é apropriado para ser utilizado em situação de MCR (*multiple correct results*)
  - Encontrar rotas entre duas cidades
  - Calcular a raiz de uma equação

Discussão sobre o NVP

- O NVP não foi projetado para prover a redundância de dados (diversidade de dados)
- Promove o *back to back testing*
  - Fornece os dados de entrada e compara as saídas de cada variante
  - Encontrar erros não coincidentes
- Interessante executar as variantes em SO para dar mais 'diversidade'
  - Em arquiteturas de hardware diferentes também
- Uso de linguagem de programação diferente tb

Discussão sobre o NVP

- Tamanho das variantes?
- Módulos/ variantes pequenos
  - Chamada frequente dos módulos de TF (baixa latência de erros, porém alto overhead)
  - Se for RcB (pouco rollback); ou menos dados corrigidos pelo 'voto'
  - Mais dados devem ser armazenados (checkpoint)
  - Mais dados devem ser votados com uma maior frequência
- A especificação deve descrever o funcionamento de todo o sistema
  - Facilita entender o problema e a integração

Discussão sobre o NVP

- O custo do NVP não é *n* vezes mais caro em relação ao *single* version. Por que?
  - Existem custos comuns (e.g. Especificação)
- NVP é normalmente executado em *n* hardwares diferentes
  - Executor reside em um dos processadores
  - Os módulos ou componentes comunicam-se via
    - RMI (*remote method invocations*)
    - Chamadas locais
- Note que erros na especificação irá replicar em todas as variantes

Discussão sobre o NVP

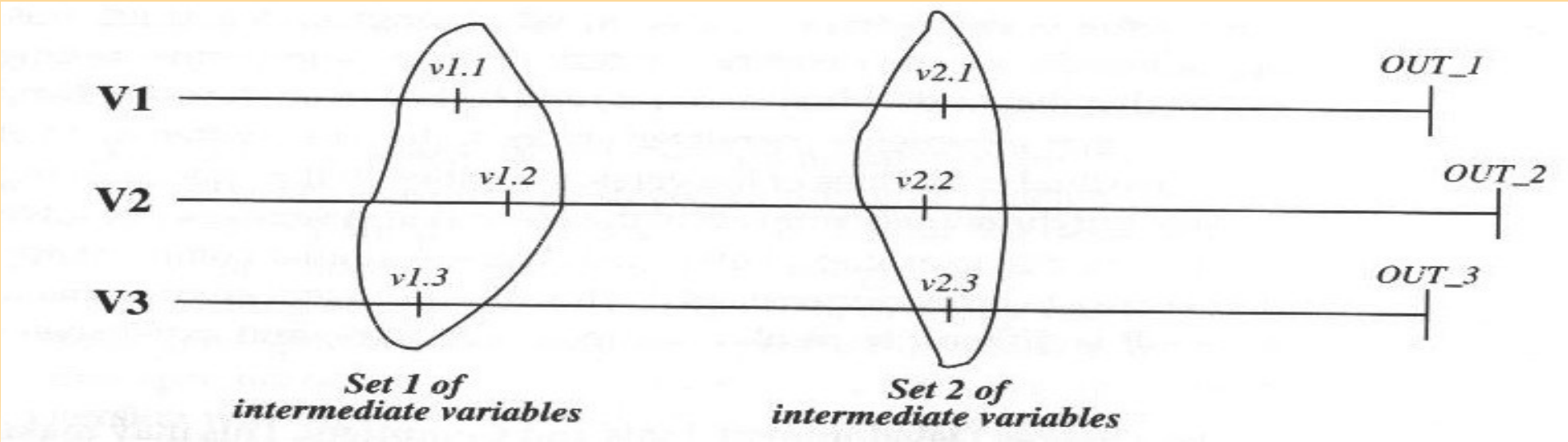
- Uso de especificações diferentes. Por que?
  - 2/3 dos erros vem de erros em especificações
- Uso de diferentes IDEs (ambiente de desenvolvimento)
  - Evitar erros 'coincidentes'
- Uma melhoria no NVP é adaptá-lo de forma que seja apropriado para a aplicação
  - Uso do *consensus voting* no DM
  - Uma generalização do *majority voting*
  - 5.0, 3.0, 5.0, 5.0, 5.0 → r* = 5.0
  - 3.0, 3.0, 2.0, 5.0, 4.0 → r* = 3.0

Discussão sobre o NVP

- Exemplos de *consensus voting*
  - 1.0, 3.0, 5.0, 3.0, 5.0 → r* = 3.0 ou 5.0 escolhido randomicamente no NVP
- Vantagens do *consensus voting*
  - Possui a mesma confiabilidade do *majority*
  - Funciona bem quando a decisão não é 'binária'
- Desvantagens
  - Complexidade é adicionada ao DM
  - Pode tomar uma decisão que não seja adequada

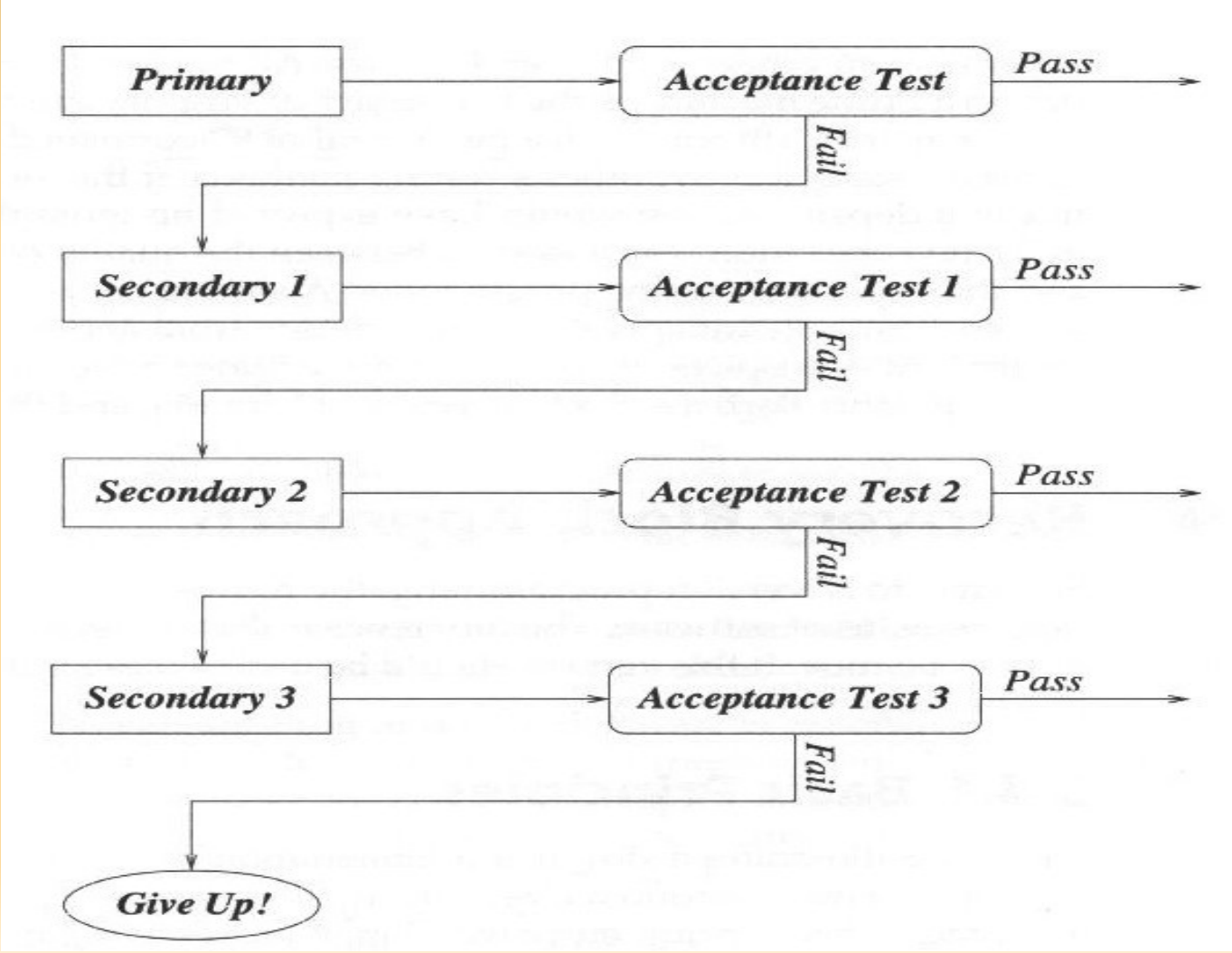
Discussão sobre o NVP

- Variáveis intermediárias em um teste back to back
  - Desenvolvedores devem criar pontos de observação intermediários
  - Facilita na identificação de falhas do que usar-se apenas um *overall output*
  - Isso pode reduzir a diversidade das variantes. Por que?
    - Restringe os programadores a isso



Revisitando a Diferença com RcV

- Estática e Dinâmica
- Estrutura do RcB com 3 variantes



Finalizando...

- Abordamos o NVP hj
- Tópico apresentado dos dois livros
- Prx aula: RcB Distribuído

Sistemas Tolerantes a Falhas

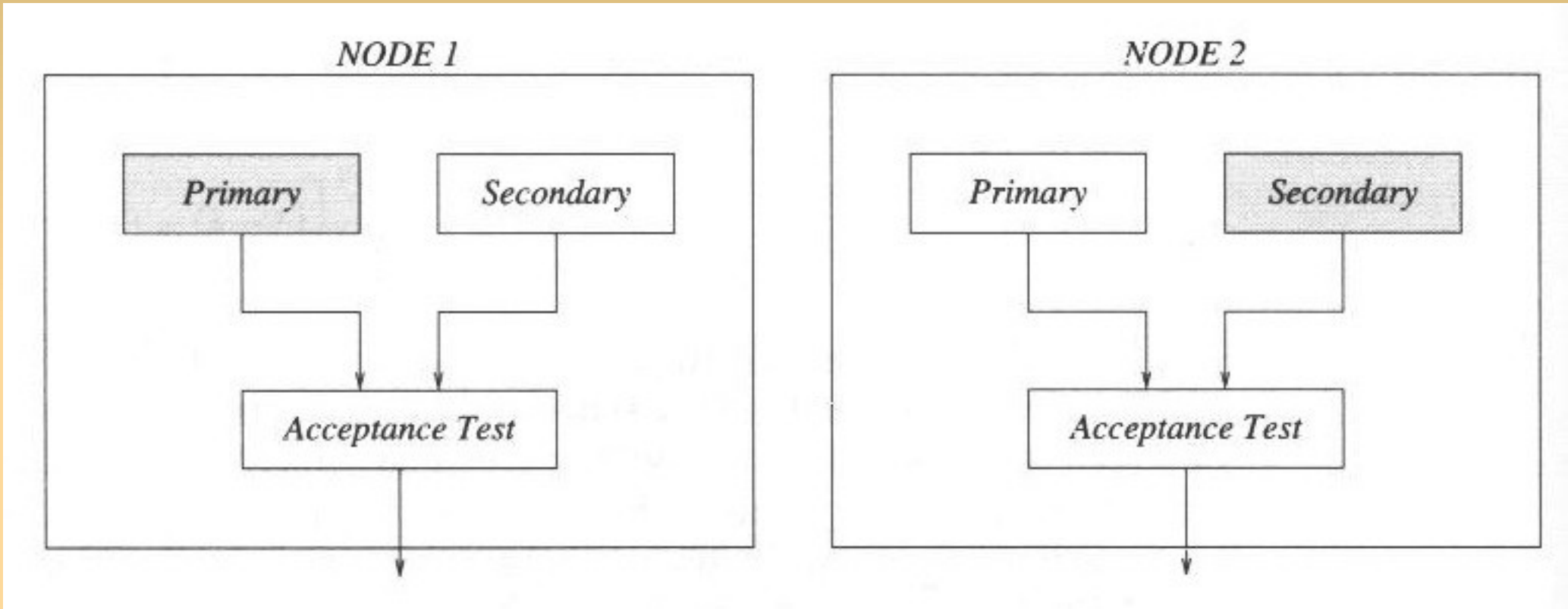
Distributed Recovery Blocks (DRBs),
Preconditions e *Postconditions*

Prof. Jó Ueyama

DRB

- Uma combinação de:
  - Processamento distribuído
  - Processamento paralelo
  - Recovery blocks
- DRB provê:
  - Tolerância a falhas no nível de hardware
  - TF no nível de software
- Apesar de usar o RcB, ele é baseado no *forward recovery*
- Uso em sistemas de tempo real
  - Por que?

Estrutura do DRB (simplificado)



DRB Simplified Code

- No código abaixo, o DRB executa o primário do nó 1 e a alternativa do nó 2
- Caso isso não suceda, então a alternativa do nó 1 e o primário do nó 2 é executado
- Em regra, isso é o que acontece com a DRB

```
run       RB1 on Node 1 (Initial Primary),
              RB2 on Node 2 (Initial Shadow)
ensure    AT on Node 1 or Node 2
by        Primary on Node 1 or Alternate on Node 2
else by   Alternate on Node 1 or Primary on Node 2
else failure exception
```

Notação e Abreviações Usadas

- A tabela abaixo apresenta as notações e as abreviações usadas

AT	Acceptance test;
Check-1	Check the AT result of the partner node with the WDT on;
*Check-1**	Check the progress of and/or AT status of the partner node;
Check-2	Check the delivery success of the partner node with the WDT on;
Status-1	Inform other node of pickup of new input;
Status-2	Inform other node of AT result;
Status-3	Inform that output was delivered to successor computing station successfully.

DRB sem Erros

Primary Node	Backup Node
Begin the computing cycle (`Cycle`).	Begin the computing cycle (`Cycle`).
Receive input data from predecessor computing station (`Input`).	Receive input data from predecessor computing station (`Input`).
Start the recovery block (`Ensure`).	Start the recovery block (`Ensure`).
Inform the backup node of pickup of new input (`Status-1` message).	Inform the primary node of pickup of new input (`Status-1` message).
Run the primary try block (`Try`).	Run the alternate try block (`Try`).
Test the primary try block's results (`AT`). The results pass the AT.	Test the alternate try block's results (`AT`). The results pass the AT.
Inform backup node of AT success (`Status-2` message).	Inform primary node of AT success (`Status-2` message).
Check if backup node is up and operating correctly. Has it taken `Status-2` actions during a preset maximum number of data processing cycles? (`Check-1*` Message) Yes, backup is OK.	Check AT result of primary node (`Check-1` message). It passed and was placed in the buffer.
Deliver result to successor computing station (`SEND`) and update local database with result.	Check to make sure the primary successfully delivered result (`Check-2` message).
—	[`Wait`]
Tell backup node that result was delivered (`Status-3` message).	Primary was successful in delivering result (No `Timeout`).
End this processing cycle.	End this processing cycle.

Falha no Primário e OK no Backup

Primary Node	Backup Node
Begin the computing cycle (`Cycle`).	Begin the computing cycle (`Cycle`).
Receive input data from predecessor computing station (`Input`).	Receive input data from predecessor computing station (`Input`).
Start the recovery block (`Ensure`).	Start the recovery block (`Ensure`).
Inform the backup node of pickup of new input (`Status-1` message).	Inform the primary node of pickup of new input (`Status-1` message).
Run the primary try block (`Try`).	Run the alternate try block (`Try`).
Test the primary try block's results (`AT`). The results fail the AT.	Test the alternate try block's results (`AT`). The results pass the AT.
Inform backup node of AT failure (`Status-2` message).	Inform primary node of AT success (`Status-2` message).
Attempt to become the backup— rollback and retry using alternate try block (on primary node) using same data on which primary try block failed (to keep the state consistent or local database up-to-date). Assume the role of backup node.	Check AT result of primary node (`Check-1` message). The primary node failed. Assume the role of primary node.
	Deliver result to successor computing station (`SEND`) and update local database with result.
Test the alternate try block's results (`AT`). The results pass the AT.	Tell primary node that result was delivered (`Status-3` message).
Inform backup node of AT success (`Status-2` message).	—
Check AT result of backup node (`Check-1` message). It passed and was placed in the buffer.	—
Check to make sure the backup node successfully delivered result (`Check-2` message).	—
Backup was successful in delivering result (No `Timeout`).	—
End this processing cycle.	End this processing cycle.

Nó Primário deixa de responder

- Este cenário parece com o do anterior
- Nenhuma mensagem de atualização (*Status*-2) é enviada ao nó de backup
- Como o nó de backup sabe sobre a falha no primário?
  - WDT
  - Expira o *timer* ligado ao envio da mensagem *Check-1*
- A execução seguiria o fluxo do caso anterior
- Se o nó de backup falhasse então o nó primário operaria sozinho
  - Assumindo que o primário não falhasse

Ambos Falham no AT

- Duas alternativas para solucionar:
- (a) reter os papéis originais, i.e., primário como primário e backup como backup
- (b) o primeiro que passar no AT assume o papel do primário
- A primeira opção é menos complexa enqt que a segunda é mais rápida
  - Qd o *retry* do primário demora mais do que o do backup

Ambos Falham no AT

Primary Node	Backup Node
Begin the computing cycle (`Cycle`).	Begin the computing cycle (`Cycle`).
Receive input data from predecessor computing station (`Input`).	Receive input data from predecessor computing station (`Input`).
Start the recovery block (`Ensure`).	Start the recovery block (`Ensure`).
Inform the backup node of pickup of new input (`Status-1` message).	Inform the primary node of pickup of new input (`Status-1` message).
Run the primary try block (`Try`).	Run the alternate try block (`Try`).
Test the primary try block's results (`AT`). The results fail the AT.	Test the alternate try block's results (`AT`). The results fail the AT.
Inform backup node of AT failure (`Status-2` message).	Inform primary node of AT failure (`Status-2` message).
Rollback and retry using alternate try block (on primary node) using same data on which primary try block failed (to keep the state consistent or local database up-to-date).	Rollback and retry using primary try block (on backup node) using same data on which alternate try block failed (to keep the state consistent or local database up-to-date).
Test the alternate try block's results (`AT`). The results pass the AT.	Test the primary try block's results (`AT`). The results pass the AT.
Inform backup node of AT success (`Status-2` message).	Inform primary node of AT success (`Status-2` message).
Check if backup node is up and operating correctly. Has it taken Status-2 actions during a preset maximum number of data processing cycles? (`Check-1*` Message) Yes, backup is OK.	Check AT result of primary node (`Check-1` message). It passed and was placed in the buffer.
Deliver result to successor computing station (`SEND`) and update local database with result.	Check to make sure the primary node successfully delivered result (`Check-2` message).
Tell backup node that result was delivered (`Status-3` message).	Primary was successful in delivering result (No `Timeout`).
End this processing cycle.	End this processing cycle.

Exemplo de DRB

- O exemplo é baseado no que foi utilizado no RcB
- Sort com falhas qd um ou mais dados < 0
- Em cada nó:
  - Sort com implementação sem erros
  - Sort com erros qd o elemento a ordenar < 0
- O AT é a comparação da soma dos elementos na entrada com a da saída
  - Mais um WDT (o *timer*)

Exemplo de DRB

Primary Node	Backup Node
Begin the computing cycle.	Begin the computing cycle.
Receive input data from predecessor computing station. Input is (8, 7, 13, –4, 17, 44). Sum the inputs for later use by AT. (*Sum of inputs* = 85.)	Receive input data from predecessor computing station. Input is (8, 7, 13, –4, 17, 44). Sum the inputs for later use by AT. (*Sum of inputs* = 85.)
Start the recovery block.	Start the recovery block.
Inform the backup node of pickup of new input.	Inform the primary node of pickup of new input.
Run the primary try block (original sort algorithm). Result = (–4, –7, –8, –13, –17, –44).	Run the alternate try block (backup sort algorithm). Result = (–4, 7, 8, 13, 17, 44).
Test the primary try block's results. *Sum of inputs* was 85; sum of results = –93, not equal. The results fail the AT.	Test the alternate try block's results. *Sum of inputs* was 85; sum of results = 85, equal. The results pass the AT.
Inform backup node of AT failure.	Inform primary node of AT success.
Attempt to become the backup—rollback and retry using alternate algorithm (on primary node) using same data on which original sort algorithm failed. Result = (–4, 7, 8, 13, 17, 44).	Check AT result of primary node. The primary node failed. Assume the role of primary node.
Test the alternate try block's (backup sort algorithm) results. *Sum of inputs* was 85; sum of results = 85, equal. The results pass the AT.	Deliver result to successor computing station and update local database with result.
Inform backup node of AT success.	Tell primary node that result was delivered.
Check AT result of backup node. It passed and was placed in the buffer.	—
Check to make sure the backup node successfully delivered result.	—
Backup was successful in delivering result.	—
End this processing cycle.	End this processing cycle.

Discussão sobre o DRB

- Carrega as vantagens e as desvantagens da diversidade e do *forward recovery*
- Algumas questões em particular do DRB
- Executado em ambientes de multiprocessamento
- Um overhead extra para executar
  - A alternativa do *shadow*
  - ambos os AT (do primário e do backup)
- Vantagem chave é o paralelismo
  - *Recovery* é realizado comunicando-se entre os nós
- A alta performance leva aos sistemas de tempo real

Discussão sobre o DRB

- O WDT promove o uso do DRB em sistemas de tempo real
- Dificuldades do DRB?
- Assim como no RcB: ATs eficientes
- O mecanismo tolera falhas tanto em SW como em HW
- *Recovery time* é mínimo: paralelismo de nós
- Não há tempo de espera, pois os nós são 'independentes'
- Importante: o *shadow* não precisa ser sofisticado qt ao *primary*

Discussão sobre o DRB

- E as desvantagens?
- Impõe o paralelismo
- Impõe que os *shadows* e os *primaries* tenham uma implementação:
  - 'semelhante'
  - 'paralela'
  - Tem que ser de duas fases (um bloco primário e um alternativo) para o *shadow* e para o *primary*
  - Necessária para permitir o *runtime recovery*
- Essencialmente: RcB usando múltiplos processadores

Discussão sobre o DRB

- E a parte de comunicação entre os processos?
  - *Remote Method Invocation* (RMI) e/ou RPC
- E o controlador?
  - Pode estar sendo executado em um outro hardware
- Estudos de performance do DRB
  - Prover confiabilidade e performance
- Entretanto, alguns dados em um determinado software executado nos radares
  - Overhead de 1,8 a 2,6 ms; pouco para um máx de 40ms
  - AT utilizou apenas 8% da CPU
  - Processamento do Backup foi considerado pequeno

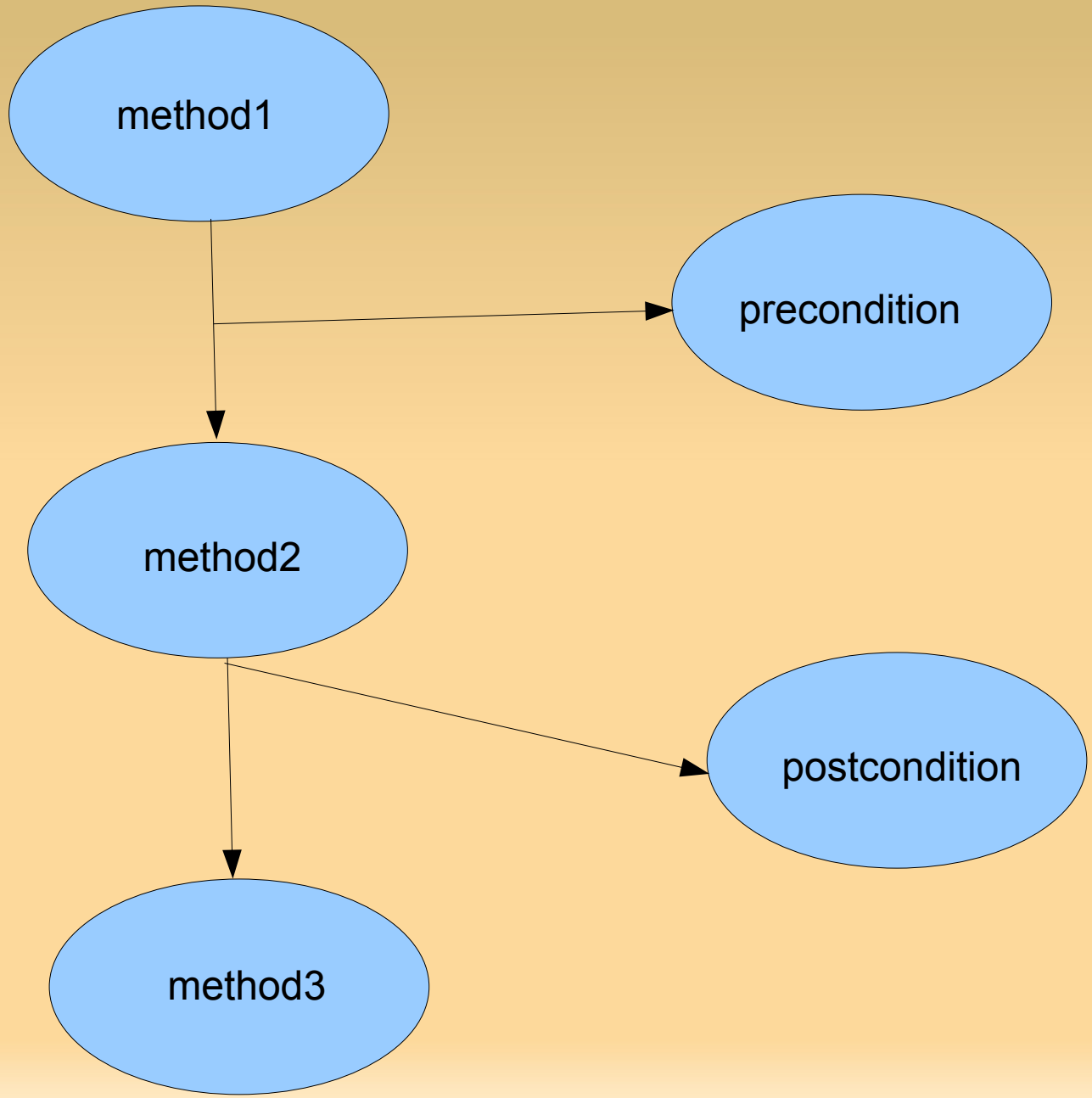
Discussão sobre o DRB

- Implementações de DRB com *n* versões
  - *n* versões executadas em paralelo;
  - em *n* unidades de processamento distintas
- Não existe *rollback* aqui. Por que?
  - O secundário é executado em paralelo
- Lembre-se: se o primário falha, então
  - O secundário vira o primário até que a situação imponha a mudança dos papéis
- Note que cópias idênticas são implementadas em ambos os nós

Preconditions e *Postconditions*

- São formas de implementarmos os ATs
- São largamente utilizados para prover confiabilidade
  - São formas de assegurarmos os contratos fora do método
- *Precondition*: é uma condição que deve ser satisfeita antes que o método seja chamado
  - Método que calcula raiz quadrada de *x*: o valor de x > 0
- *Postcondition*: condição que deve ser satisfeita quando saímos do método
  - Método do ajuste do horário para x: o valor final tem que ser um horário válido

Preconditions e Postconditions



Preconditions e *Postconditions*

- Por que não implementar o mecanismo na própria classe e/ou componentes?
  - *Separtion of concerns*
  - Permite utilizar classes e/ou componentes genéricos

Finalizando...

- Abordamos o DRB, *Precondition* e o *Postcondition*
- Esta aula foi baseada nos dois livros
- Prx aula: *N Self Checking Programming*