Desenvolvimento de Sistemas Tolerantes a Falhas

Confiança de software

- Em geral, os usuários de um sistema de software esperam ele seja confiável
  - Para aplicações não-críticas, podem estar dispostos a aceitar algumas falhas
- Algumas aplicações, contudo, têm requisitos muito altos de confiabilidade
  - Exigem técnicas específicas de engenharia de software

Para atingir a confiança

- Prevenção de falhas
  - O sistema é desenvolvido de modo que provadamente atenda sua especificação
- Remoção de falhas
  - Técnicas de verificação e de validação são usadas para descobrir e remover falhas em um sistema antes que seja entregue.
- Tolerância a falhas
  - O sistema é projetado de forma que as falhas no software entregue não resultem em defeitos em tempo de execução

Para atingir a confiança

- Prevenção de falhas
  - O sistema é desenvolvido de modo que provadamente atenda sua especificação
- Remoção de falhas
  - Técnicas de verificação e de validação são usadas para descobrir e remover falhas em um sistema antes que seja entregue.
- Tolerância a falhas
  - O sistema é projetado de forma que as falhas no software entregue não resultem em defeitos em tempo de execução

Diversidade e redundância

- Redundância
  - Parte do sistema ou de suas ações que não seria necessária se falhas não existissem
  - Exs.: componente duplicado, reinicialização
- Diversidade
  - Prover a mesma funcionalidade de maneiras diferentes para que eles não falhem de forma análoga
- A adição de diversidade e de redundância aumenta a complexidade
  - Simplicidade e V & V extensivas podem ser uma rota mais eficiente para a confiança de software

Exemplos de diversidade e redundância

- Redundância. Onde a disponibilidade é crítica (por exemplo, em sistemas de *e-commerce*), as empresas normalmente mantêm servidores de backup e chaveiam automaticamente caso ocorram falhas.
- Diversidade. Para fornecer resistência contra ataques externos, diferentes servidores podem ser implementados com o uso de sistemas operacionais diferentes (por exemplo, Windows, Mac OS e Linux).

Software livre de falhas

- Os métodos atuais de engenharia de software permitem a produção de software livre de defeitos
  - Para sistemas relativamente pequenos.
- Significa que o software atende à especificação
  - NÃO significa que sempre executará corretamente
- O custo de produção de software livre de falhas é muito alto.
  - É mais barato aceitar falhas e pagar por suas conseqüências

Custos de remoção de defeitos

Figura 20.1
Custos crescentes de remoção de defeitos residuais.

Processos confiáveis

- Para assegurar um número mínimo de defeitos de software, é importante ter um processo de software bem definido e repetível.
- Um processo bem definido e repetível é aquele que não depende inteiramente de habilidades individuais; isto é, podem ser realizados por pessoas diferentes;
- Para detecção de defeitos, é claro que as atividades de processo devem incluir um esforço significativo dedicado à verificação e à validação.

Atividades que ajudam na remoção de falhas

- Inspeções de requisitos
  - Descoberta de problemas com a especificação
- Gerenciamento de requisitos
  - Acompanhamento de mudanças no projeto e implementação
- Verificação de modelos
  - Análise automática de modelos
- Inspeções de projeto e de codificação
  - Listas de defeitos comuns para descoberta e remoção deles

Atividades que ajudam na remoção de falhas

- Análise estática
  - Análise de programas
- Planejamento e gerenciamento de teste
  - Cobertura e rastreabilidade entre testes e requisitos
- Gerenciamento de configuração
  - Inclusão correta de componentes no sistema
- Arquiteturas adequadas
  - Remoção localizada de erros, facilidade de introdução de

Tolerância a Falhas

- Sistemas críticos devem ser tolerantes a falhas
- Tolerância a falhas significa que o sistema pode continuar em operação apesar da falha do software
- Complementar a técnicas para remover falhas
- Abordagem **muito** usada na prática
  - Microsoft Office, Windows, OpenOffice, Firefox, Opera
- Um sistema é tolerante a falhas com relação a um dado modelo de falhas
- Falha, erro, defeito

Etapas de tolerância a defeitos

Detecção de erros
O sistema deve detectar se um erro ocorreu (um estado incorreto de sistema)

Avaliação de danos
As partes do estado de sistema afetadas pelo erro devem ser detectadas

Recuperação de erros
O sistema deve ir para um estado 'seguro' estável

Reparação de erros
O sistema pode ser modificado para evitar recorrência da falha

Tudo isso é feito em tempo de execução!

Detecção de erros

- O primeiro estágio de tolerância a falhas é detectar se um erro ocorreu
- Envolve a definição de restrições que devem ser mantidas em todos os estados legais, e a verificação do estado contra essas restrições

Restrições de estado de uma bomba de insulina

Figura 20.6
Restrições de estado que se aplicam à bomba de insulina.

```
// A dose de insulina a ser liberada deve ser sempre maior
// do que zero e menor do que alguma dose máxima única definida

insulin_dose >= 0 & insulin_dose <= insulin_reservoir_contents

// A quantidade total de insulina liberada em um dia deve ser menor
// do que ou igual à dose máxima diária definida

cumulative_dose <= maximum_daily_dose
```


Avaliação de danos

- Analisa o estado do sistema para estimar a extensão da corrupção causada por um erro
- Deve verificar quais partes do estado do sistema foram afetadas pela erro
- Geralmente, é baseada em 'funções validadas', que podem ser aplicada aos elementos de estado para avaliar se seu valor está dentro de uma faixa permitida.

Figura 20.8
Classe de vetor com avaliação de danos.

```
class RobustArray {

    // Verifica se todos os objetos em um vetor de objetos
    // atendem a alguma restrição definida

    boolean [] checkState ;
    CheckableObject [] theRobustArray ;

    RobustArray (CheckableObject [] theArray)
    {
        checkState = new boolean [theArray.length] ;
        theRobustArray = theArray ;
    } //RobustArray

    public void assessDamage () throws ArrayDamagedException
    {
        boolean hasBeenDamaged = false ;

        for (int i= 0; i <this.theRobustArray.length ; i ++)
        {
            if (! theRobustArray [i].check ())
                {
                    checkState [i] = true ;
                    hasBeenDamaged = true ;
                }
            else
                checkState [i] = false ;
        }
        if (hasBeenDamaged)
            throw new ArrayDamagedException () ;
    } //assessDamage

} // RobustArray
```


Recuperação de erros

- Recuperação por avanço (*forward error recovery)*
  - Levar o estado do sistema para um novo estado livre de erros
- Recuperação por retrocesso (*backward error recovery)*
  - Restaurar um estado anterior, livre de erros, do sistema
- A recuperação por avanço é específica de aplicação
- A recuperação por retrocesso de erros é mais simples e genérica
  - Não se aplica a todos os casos, porém

Recuperação por avanço

- Corrupção de dados codificados
  - Adicionam redundância aos dados codificados -- dados redundantes podem ser usados para reparação de dados corrompidos
- Ponteiros redundantes
  - Quando ponteiros redundantes são incluídos em estruturas de dados uma lista ou repositório de arquivos corrompido pode ser reconstruída se um número suficiente de ponteiros não estão corrompidos.
  - Freqüentemente usados para reparação de banco de dados e sistemas de arquivos.
- Tratamento de exceções
  - Ênfase na estruturação

Recuperação por retrocesso

- Transações são um método popular
  - Mudanças não são aplicadas até que a computação seja completada
  - Se um erro ocorre, o sistema é deixado no estado anterior ao da transação.
- Checkpoints locais
  - Usados pelo MS Office e pelo OpenOffice.
- Checkpoints distribuídos periódicos
  - Baseados em *logs*

Arquiteturas tolerantes a falhas

- Técnicas de V & V não são capazes de ajudar a prever interações entre o hardware e o software.
- Mau entendimento dos requisitos pode significar que as verificações são incorretas ou incompletas
- Se requisitos de confiabilidade são críticos, pode ser usada uma arquitetura específica para apoiar tolerância a falhas
- Podem tolerar falhas de hardware e/ou de software.

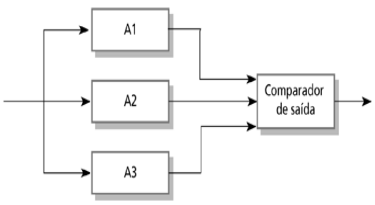
Tolerância a falhas de hardware

- Depende da redundância modular tripla (TMR)
- Três componentes idênticos replicados recebem a mesma entrada e têm suas saídas comparadas
- Se uma saída é diferente, ela é ignorada e uma falha do componente é simulada.
- Premissas básica:
  - Falhas de hardware decorrem de falhas de componentes (e.g. por desgaste natural) e não de projeto
  - Falhas são exceções e falhas simultâneas são raras

Confiabilidade de hardware com TMR

Figura 20.10

Redundância modular tripla para lidar com falha de hardware.



TMR e Software

- As suposições básicas da TMR não valem para sistemas de software
  - Não é possível simplesmente replicar os mesmos componentes quando eles têm falhas de projeto em comum;
  - Falhas simultâneas de componentes seriam, portanto, virtualmente inevitáveis.
- Para ser tolerantes a falhas, sistemas de sofware devem incluir diversidade

Diversidade de projeto

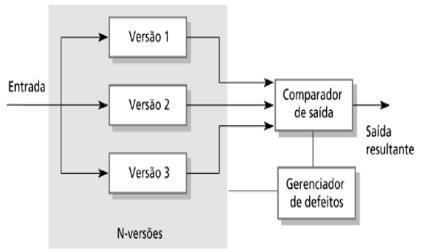
- Versões diferentes do sistema são projetadas e implementadas de maneiras diferentes.
  - Podem apresentar modos de falha diferentes.
- Abordagens diferentes para projeto
  - Implementação em linguagens de programação diferentes;
  - Uso de ferramentas e ambientes de desenvolvimento diferentes;
  - Execução em diferentes sistemas operacionais
  - Algoritmos e escolhas de projeto diferentes

Abordagens para Diversidade de Projeto

- Programação de N-versões
  - A mesma especificação é implementada em uma série de versões diferentes por equipes diferentes
  - Todas as versões calculam simutâneamente e a saída da maioria é selecionada usando um sistema de votação.
  - Essa é a abordagem mais comumente usada, por exemplo, em muitos modelos da aeronave comercial Airbus.
- Blocos de recuperação
  - Uma série de versões explicitamente diferentes da mesma especificação são escritas e executadas em seqüência.
  - Um teste de aceitação é usado para selecionar a saída a ser transmitida.

Programação em N-versões

Figura 20.11
Programação em N-versões.



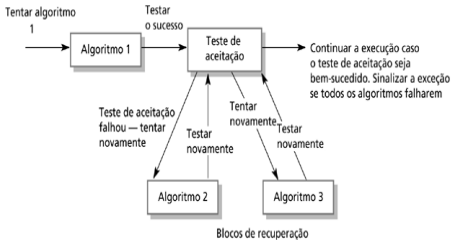
Programação em N-versões

- As versões diferentes de sistema são projetadas e implementadas por equipes diferentes.
- Admite-se que existe uma baixa probabilidade dessas equipes cometerem os mesmos erros
- Existe evidência empírica [1] de que equipes
  - cometem com freqüência erros de interpretação de especificações da mesma maneira e
  - escolhem os mesmos algoritmos em seus sistemas.

[1] AN EXPERIMENTAL EVALUATION OF THE ASSUMPTION OF INDEPENDENCE IN MULTI-VERSION PROGRAMMING

Blocos de Recuperação

Figura 20.12
Blocos de recuperação.



Blocos de recuperação

- Forçam que vários algoritmos diferentes sejam usados
  - Reduzem a probabilidade de erros comuns.
- Exigem um teste de aceitação
  - Difícil de elaborar de forma independente do algoritmo
- Não é sempre aplicável a sistemas de tempo real, devido à sua operação seqüencial
- Existe evidência de que a programação em N-versões também não é!

Problemas com diversidade de projeto

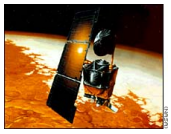
- Equipes não são culturalmente diversas e tendem a resolver os problemas da mesma maneira.
- Erros característicos:
  - Equipes diferentes comentem os mesmos erros.
  - Algumas partes de uma implementação são mais difíceis que outras
    - Equipes tendem a cometer erros no mesmo lugar;
  - Falhas na especificação;
    - São refletidas em todas as implementações;
- Esses problemas podem ser mitigados pelo uso de representações múltiplas da especificação.

Exemplos do que pode acontecer

Ariane 5

Míssil Patriot

Mars Climate Orbiter



Mais informações:
http://depend.cs.uni-sb.de/index.php?id=336
http://www.cis.gsu.edu/~mmoore/CIS3300/handouts/SciAmSept1994.html