Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Pós-Graduação em Ciência da Computação

DIMINUIÇÃO DA INTRUSÃO DO TESTE DE SOFTWARE EM PROGRAMAS PARALELOS

por

Leonardo Albernaz Amaral

Dissertação apresentada como requisito parcial para obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Eduardo Augusto Bezerra

Porto Alegre, janeiro de 2006.

Dados Internacionais de Catalogação na Publicação (CIP)

A485d Amaral, Leonardo Albernaz
Diminuição da intrusão do teste de softwares em programas paralelos / Leonardo Albernaz Amaral. – Porto Alegre, 2006.
111 f. : il.

Dissertação (Mestrado) – Fac. de Informática, PUCRS, 2006.

1. Processamento Paralelo. 2. Engenharia de Software. 3. Software – Técnicas de Avaliação. 4. Redes de Computadores. I. Título.

CDD 005.1

Ficha Catalográfica elaborada pelo
Setor de Processamento Técnico da BC-PUCRS

Campus Central
Av. Ipiranga, 6681 – prédio 16 – CEP 90619-900
Porto Alegre – RS – Brasil
Fone: +55 (51) 3320-3544 – Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Reconhecido pelo Parecer No. 930/98.C.N.E. Homologação Publicada no D.O.U. de 30/12/98.

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Diminuição da Intrusão do Teste de Software em Programas Paralelos***", apresentada por Leonardo Albernaz Amaral, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Confiabilidade de Sistemas, aprovada em 17/01/2006 pela Comissão Examinadora:

Prof. Dr. Eduardo Augusto Bezerra – PPGCC/PUCRS
Orientador

Prof. Dr. Avelino Francisco Zorzo – PPGCC/PUCRS

Prof. Dr. Flávio Moreira de Oliveira – FACIN/PUCRS

Prof. Dr. Raul Ceretta Nunes – UFSM

Homologada em 08/03/06, conforme Ata No 005 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

AV. IPIRANGA, 6681 - PRÉDIO 16/SALA 106
90619-900 - PORTO ALEGRE -RS -BRASIL
TELEFONE: 55 51 33203611 FAX 55 51 3320.3621
http: //www.pucrs.br/inf/pos e-mail: ppgcc@inf.pucrs.br

Dedico esta dissertação aos meus pais, Paulo e Deolinda, a minha irmã Alessandra e a minha namorada Raquel.

AGRADECIMENTOS

O Mestrado é uma escolha por onde passamos e dedicamos intensos dois anos de nossas vidas na busca de um conhecimento, uma informação, uma sabedoria, uma aprovação, algo capaz de transformar nossas vidas, mudando não apenas a nossa maneira de pensar, mas também a maneira com a qual lidamos com nossas próprias limitações. Por esses motivos e por muitos outros que eu gostaria de agradecer a todos aqueles que de alguma forma me ajudaram nessa conquista, mas principalmente:

Aos meus pais, Paulo e Deolinda, por tudo o que sempre fizeram por mim e por tudo o que sempre irão fazer.

A minha irmã Alessandra, por sempre estar do meu lado me incentivando, ajudando e muitas vezes fazendo bem mais do que o papel de irmã.

A minha amada e namorada de muitos anos, Raquel, onde sempre pude e espero poder contar, além de muito amor e dedicação, também com todo o seu companheirismo, amizade e respeito.

Ao meu orientador Prof. Eduardo Bezerra, uma pessoa que me acolheu e que se tornou um grande amigo. Agradeço a ele pela confiança, sabedoria, paciência e principalmente por todas as oportunidades que me proporcionou. Muitas vezes o seu otimismo e bom humor foi um grande incentivo ao meu trabalho.

Aos pais da minha namorada, Romeu e Márcia, por muitas conversas sábias e outros tantos momentos que passamos durante esses anos de convivência. Tenho certeza que aprendo muito com eles.

Ao meu co-orientador, Prof. Flávio Moreira de Oliveira, pela sua amizade e sabedoria, e também ao pessoal do CPTS/HP, onde pude ter bons momentos.

Aos colegas de trabalho, principalmente o Pedro e o Mateus (CAP/HP), onde juntamente com o Prof. Luiz Gustavo (CAP/HP), tiveram extrema importância para o desenvolvimento do meu trabalho.

E a HP Brasil, pelo financiamento da minha bolsa de estudos.

MUITO OBRIGADO!

RESUMO

Nesse trabalho é apresentada uma estratégia para diminuir a intrusão do teste de software em programas paralelos baseados em troca de mensagens. Para isso, um ambiente de teste foi desenvolvido utilizando técnicas de teste de software funcional e abordagens de depuração. O ambiente, que utiliza Java como linguagem de programação e MPI como biblioteca para troca de mensagens, baseia-se na idéia de utilizar Rede de Autômatos Estocásticos (SAN) para a representação do modelo comportamental da aplicação e, com isso, criar casos de teste que exercitem a aplicação paralela na busca por falhas de comunicação entre os processos. Essas falhas são identificadas pelos módulos de monitoração e análise *on-line*, que observam a execução da aplicação, verificando inconsistências entre os estados atingidos e os estados esperados do modelo. Para a diminuição da intrusão foi dada ênfase tanto para a definição e geração de casos de teste, quanto para as abordagens utilizadas no *engine* de teste nas etapas de monitoração e análise. Busca-se com essa estratégia, validar as abordagens utilizadas no processo de teste e identificar eventuais problemas.

ABSTRACT

This work introduces a strategy aiming the reduction of the intrusion (probe-effect mitigation) resulting from software testing activities in message passing parallel programs. In order to accomplish this goal, a test environment has been developed, based on functional software testing techniques and debugging approaches. The environment, which makes use of Java as the programming language and MPI as the library for message exchange, is based on the use of Stochastic Automata Network (SAN) for the application behavioural model representation. Test cases are created from this behavioural model to stimulate parallel programs, seeking for inter-process communication errors. These faults are identified by monitoring and analysis modules, using on-line approaches, that observe the application's execution looking for inconsistencies between reached and expected states in the model. In order to define a less intrusive test strategy, emphasis has been given not only to the definition and generation of test cases, but also to the approaches employed in the test engine for the monitoring and analysis stages. It is expected that this strategy would help in the validation of the testing process approaches, and also in the identification of eventual problems.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS E ABREVIATURAS

Y2K	Referência ao ano 2000
STAGE	State-based Test Generator
CASE	Computer Aided Software Engineering
FSM	Finite State Machine
MPI	Message Passing Interface
PVM	Parallel Virtual Machine
P2D2	Portable Parallel Distributed Debugger
PDBG	Process-level Debugger for Parallel and Distributed Programs
PDT	Parallel Debugging Tool
GBD	GNU Debugger
DDBG	Distributed Debugger

Capítulo 1

INTRODUÇÃO

1.1 CONTEXTO

Com o passar dos anos o computador se tornou uma das ferramentas mais importantes já desenvolvidas pela humanidade. Ele atualmente está inserido na vida de muitas pessoas, provendo inúmeras facilidades que automatizam e aceleram tarefas tediosas do dia-a-dia, fazendo com que seus usuários consigam tempo para realizar atividades consideradas mais produtivas e interessantes.

O universo de utilização dos computadores alcançou lugares onde há poucos anos atrás nem se imaginam o uso de tal tecnologia. A disponibilidade da Internet em boa parte das residências e empresas ao redor do mundo, os telefones celulares, assim como diversos outros dispositivos móveis e de entretenimento, providos de um grande poder de processamento, são exemplos desse crescimento.

Muitas pessoas acreditam que existe um número excessivo de computadores e que alguns são usados de maneira desnecessária, e até mesmo incorreta, algo que muitas vezes põe em dúvida a funcionalidade dessas máquinas. Essa desconfiança se agravou, principalmente, durante o Bug do Milênio (*Y2K problem*), e continua influenciada pela disposição atual do mercado. Uma forte indicação disso é a crescente utilização de softwares embarcados e tecnologias sem fio em dispositivos eletroeletrônicos de última geração, o que cultiva nos usuários dessas tecnologias, a incerteza de onde toda essa explosão tecnológica irá parar.

Além dessas tendências atuais e futuras de mercado, a computação é aplicada também em áreas mais tradicionais da ciência e engenharia. A simulação é uma delas, e compõe o campo científico e metodológico ao lado de categorias teóricas e experimentais, sendo através desses domínios mais científicos que a computação torna-se indispensável na solução de grandes desafios.

Até o presente momento, diversos problemas no campo científico ainda não foram solucionados, ou foram resolvidos de maneira parcial, devido a limites no poder computacional existente. Uma visível tendência, é que essa demanda por alto desempenho computacional nunca termine, e que os problemas sempre venham a persistir.

Esse grupo de problemas também chamado de “grandes desafios” afeta áreas como: pesquisas climáticas, análises químicas, estudos do espaço, entre outras [2]. Conseqüentemente há uma crescente demanda por um maior poder computacional, fazendo com que pesquisas sejam estimuladas na busca por alto desempenho, objetivando desenvolver computadores com maior capacidade de processamento e memória, mais espaço em disco, e principalmente com dimensões reduzidas.

Diversas arquiteturas de computadores estão disponíveis atualmente para os mais variados tipos de aplicações. A mais utilizada é a arquitetura seqüencial tradicional que foi idealizada por *Von Neumann* [2]. Desde sua criação até hoje, muita coisa mudou nessa arquitetura: os processadores adquiriram um maior processamento; houve um aumento na capacidade de memória; e a transmissão de dados recebeu barramentos mais rápidos, tornando-a mais eficiente. Tudo isso acompanhado por uma diminuição do custo, o que tornou a tecnologia computacional mais acessível.

No entanto, atualmente está se tornando cada vez mais difícil aumentar o desempenho de máquinas baseadas nesse modelo de *Von Neumann*. Aplicações que necessitam de um grande poder computacional buscam suas soluções em computadores que utilizam outras arquiteturas, como as arquiteturas paralelas. Contudo, melhorar o desempenho não depende somente do uso de dispositivos de hardware mais rápidos e seguros, depende também de uma melhoria na arquitetura dos computadores e nas técnicas de processamento. Nesse ponto, há uma grande diferença entre usuários que estão interessados em eficiência e confiabilidade, e usuários de tecnologias tradicionais do dia-a-dia, e que buscam apenas solucionar problemas de maneira eficaz. Os usuários que buscam alto desempenho sabem como programar suas aplicações de maneira mais eficiente para a solução de determinados problemas. Além disso, as máquinas usadas por esses usuários são capazes de trabalhar dias, semanas ou até mesmo meses na solução de um problema em particular.

Desde que surgiu há 20 anos atrás a computação paralela vem permitindo que problemas complexos sejam computacionalmente solucionados, e aplicações de alto desempenho sejam desenvolvidas. A utilização desses computadores de larga escala, juntamente com suas aplicações, introduzem problemas que são estabelecidos principalmente devido a sua complexidade, ao grande número de processos envolvidos e também pela quantidade de dados a serem processados. Muitas dessas complexidades adicionais são inseridas devido ao fato de que programar aplicações paralelas é mais complicado quando comparado com programas seqüenciais.

Muitos desses problemas são observados durante o ciclo de vida do desenvolvimento de programas paralelos, especialmente durante as fases de teste, depuração e ajustes de desempenho *(tuning phase)*. O objetivo dessas fases é detectar falhas e corrigir erros críticos e gargalos de desempenho, aumentando a confiabilidade e eficiência do código da aplicação, e resultando em uma aplicação com maior qualidade. Esse propósito é conseguido basicamente através da análise dos estados do programa durante sua execução e da interpretação desses resultados. Entretanto, a quantidade de dados que descreve esses estados depende do número

de processos paralelos envolvidos. Conseqüentemente, o tamanho dos resultados depende da quantidade de estados monitorados e também do tempo total da execução da aplicação. Portanto, devido a essa tendência em gerar grandes quantidades de dados a serem analisados é que muitas vezes se excede o limite de memória disponível, limitando e dificultando assim as atividades de teste e depuração. Outra conseqüência, é que essas atividades de análise e monitoração podem influenciar de maneira negativa o comportamento do programa testado, alterando a ordem e o tempo de execução dos eventos.

Além desses problemas, o teste e a depuração de programas paralelos tornam-se bem mais complicados do que em programas seqüenciais, principalmente por causa do não-determinísmo e seus problemas relacionados. Muitos pesquisadores tentam estender as técnicas e abordagens utilizadas no teste e depuração de programas seqüenciais para serem aplicadas em programas paralelos. Entretanto, dada a complexidade desse tipo de arquitetura de software, inúmeros problemas ainda persistem.

Nesse trabalho definiu-se uma estratégia para a diminuição da intrusão do teste de software em programas paralelos, onde um ambiente de teste foi desenvolvido para validar as técnicas propostas. Espera-se com essa estratégia conseguir criar uma abordagem que seja menos intrusiva no comportamento da aplicação em teste. No entanto, para isso, a intrusão tem que ser diminuída não apenas nas etapas de teste, mas principalmente na técnica de monitoração a ser utilizada. **Dessa forma, foi definida uma estratégia híbrida utilizando tanto técnicas de teste de software funcional quanto técnicas de depuração e monitoração de programas paralelos.**

O ambiente de teste desenvolvido é composto basicamente por três módulos: um módulo para geração automática de casos de teste e scripts, um módulo de análise, e um módulo de monitoração (*on-line*) e execução da aplicação em teste. Os casos de teste são gerados automaticamente através de um modelo abstrato da aplicação (modelo de uso). O formalismo utilizado para essa modelagem é SAN (*Stochastic Automata Network*) [35], ou Rede de Autômatos Estocásticos, o qual através da associação de uma distribuição de probabilidade nas transições entre os estados do modelo, passa a descrever o uso operacional do software, ou o comportamento esperado.

Além do ambiente implementado, foram definidos também: um processo de teste, que estipula as etapas e técnicas utilizadas; e um ciclo de teste, que especifica as abordagens utilizadas em cada etapa do processo.

1.2 MOTIVAÇÃO

A principal motivação desse trabalho é pesquisar métodos e técnicas importantes relacionados ao teste e a depuração de programas paralelos e, com isso, adquirir um conhecimento científico capaz de nos permitir propor uma nova estratégia de teste de software, focando as

abordagens utilizadas no teste de software funcional, assim como na diminuição da intrusão dessas etapas de teste.

Outra forte motivação é a carência de ferramentas de teste para ambientes paralelos, algo que é sentido nas bibliografias da área. No entanto, nos sentimos motivados também, e principalmente, pelo fato das pessoas envolvidas nesse projeto fazerem parte de centros de pesquisa relacionados tanto ao teste de software (CPTS[1]) quanto ao processamento paralelo e distribuído (CAP[2]), algo que promove a integração e a extensão de projetos. Como é o caso do STAGE [1], uma ferramenta de teste desenvolvida no CPTS, e que nesse trabalho permite adaptar as suas funcionalidades de geração automática de casos de teste para interfaces de aplicações web, a fim de gerar também, casos de teste para aplicações paralelas não-determinísticas.

1.3 OBJETIVOS

O objetivo principal desse trabalho **é propor uma estratégia para diminuir a intrusão do teste de software em programas paralelos**. Para isso, é necessária a definição de um processo de teste e a construção de um ambiente que implemente as técnicas utilizadas nesse processo. Como objetivo secundário, busca-se validar as funcionalidades desse ambiente e levantar as dificuldades que eventualmente impossibilitem essa validação.

1.4 ORGANIZAÇÃO

Este trabalho está organizado da seguinte maneira:

- O **Capítulo dois** introduz a área de aplicação dessa dissertação, que é a **computação paralela**. Nele é feita uma revisão da história dos computadores, dando ênfase às arquiteturas paralelas, e descrevendo as razões para o estabelecimento de arquiteturas de alto desempenho, onde são citados os termos “supercomputador” e “grandes desafios”.
- O **Capítulo três** apresenta o domínio do problema, que é o **teste e a depuração**. É feita então uma revisão dos principais conceitos sobre teste de software e depuração, contextualizando as suas atividades no ciclo de desenvolvimento de software. No decorrer do capítulo, é apresentada uma classificação das principais nomenclaturas envolvidas. Em seguida, são apresentadas às fases do teste e as principais técnicas e critérios de teste, onde é dada uma atenção especial para o teste de software funcional, destacando os modelos de uso e os critérios estatísticos de teste.

[1] CPTS: Centro de Pesquisa em Teste de Software – Cooperação PUCRS/HP-Brasil.
[2] CAP: Centro de Pesquisa em Aplicações Paralelas – Cooperação PUCRS/HP-Brasil.

- O **Capítulo quatro** apresenta os principais **trabalhos relacionados** com o domínio do problema. São apresentados em mais detalhes os ciclos de teste e depuração, tanto para programas seqüenciais quanto para programas paralelos. Inicia-se com uma abordagem tradicional sobre teste, e no decorrer do capítulo são introduzidos alguns problemas e dificuldades encontradas ao se analisar programas paralelos. Dessa forma, esse capítulo apresenta o referencial teórico da dissertação.
- No **Capítulo cinco** é apresentado o ambiente desenvolvido para a validação da estratégia proposta (objetivos do trabalho). Nele é apresentado o processo de teste, o qual define as etapas e técnicas utilizadas, além de um ciclo de teste, que oferece mais detalhes às abordagens de cada etapa envolvida. São apresentadas também: as funcionalidades do ambiente de teste; e os resultados e dificuldades encontradas na tentativa de validar essas funcionalidades.
- No **Capítulo seis**, a estratégia proposta para a diminuição da intrusão do teste de software em programas paralelos é apresentada. Experimentos são discutidos e seus resultados avaliados, visando medidas de desempenho que comparem a nossa abordagem de monitoração com uma outra ferramenta de monitoração comercial.
- Por fim, o **Capítulo sete** apresenta as **conclusões** e **trabalhos futuros**.

Capítulo 2

ÁREA DE APLICAÇÃO: COMPUTAÇÃO PARALELA

Este capítulo introduz o ambiente de trabalho descrito nessa dissertação. O domínio do problema faz parte da computação paralela, e é importante para descrever as razões para o estabelecimento de arquiteturas de alto desempenho, o que é indispensável para a solução de grandes problemas ligados à ciência e à engenharia. São apresentados os termos "supercomputador" e "grandes desafios", assim como uma visão geral da situação atual da área de computação paralela de alto desempenho.

2.1 GERAÇÃO DOS COMPUTADORES

Desde a invenção dos primeiros computadores que o poder computacional oferecido por essas máquinas não satisfaz as exigências de engenheiros, cientistas e usuários que necessitam de alto desempenho para a solução de problemas complexos. Conseqüentemente, essa necessidade se tornou uma busca incansável por desempenho, resultando em uma grande evolução na história dos computadores.

Basicamente já se passaram quatro séculos de tentativas na construção de máquinas computacionais. Muitos projetos tiveram sucesso, entretanto, diversos nem saíram do papel. Mesmo assim, todos os esforços foram válidos, pois algo que no passado era apenas um sonho, hoje é a realidade, a qual tornou os computadores indispensáveis à vida das pessoas.

Os principais avanços na computação vieram com a criação das primeiras máquinas de calcular e com os primeiros computadores, os quais vêm sendo desenvolvidos e aperfeiçoados ao longo dos anos. Uma das mais importantes investidas na área computacional, e que merece registro histórico, foi a do inglês *Charles Babbage* por volta de 1830. Ele projetou dois computadores, o *Difference Engine* (Dispositivo Diferencial) e o *Analytical Engine* (Dispositivo Analítico), sendo o primeiro para realizar computações matemáticas mais especificas, e o segundo para propósitos matemáticos gerais [2]. Ambos representaram grandes avanços científicos para a época, embora não tenham sido implementados. Outro

importante avanço, e que foi uma das primeiras tentativas de construção de computadores eletrônicos, ocorreu por volta de 1930, por *John Atanasoff.* Seu computador era baseado em válvulas, e tinha o propósito de resolver equações lineares. Entretanto, os primeiros computadores de propósitos gerais, foram provavelmente o COLOSSUS e o ENIAC, construídos entre 1940 e 1946 [2]. Parte da motivação do *ENIAC*, e analogamente à primeira máquina de *Babbage*, foi a necessidade de construir tabelas de forma automática, algo que despertou o interesse do exército americano. Fisicamente, o *ENIAC* era uma máquina enorme que pesava trinta toneladas e empregava cerca de 1800 válvulas [2].

Com o avanço das pesquisas e o conseqüente desenvolvimento tecnológico, houve grandes modificações nos computadores ao longo do tempo. Como reflexo dessa evolução, a tecnologia e os estilos usados na construção e programação de computadores formaram várias gerações, conforme a Tabela 1.

Tabela 1: Geração dos Computadores

Geração/Período	Tecnologia e Arquitetura	Software e Sistema Operacional	Sistema Representativo
Primeira (1946-1956)	Válvulas e memórias de tubos catódicos	Linguagem de máquina e assembly	COLOSSUS, ENIAC, IBM 701, Princeton IAS
Segunda (1956-1967)	Transistores, núcleos de ferrite, discos magnéticos, barramentos de I/O	Algol e Fortran, compiladores, processamento em lotes	IBM 7030, CDC 1604, Univac LARC
Terceira (1967-1978)	CI (Circuitos Integrados) (SSI)	Linguagem C, multiprogramação, time-sharing	PDP-11, IBM 360/370, CDC 6600
Quarta (1978-1989)	Microprocessadores VLSI, multiprocessadores	Multiprocessamento, compiladores paralelos, bibliotecas de troca de mensagens	IBM PC, VAX 9000, Cray X/MP
Quinta (1990-Atualmente)	Circuitos ULSI, computadores paralelos escaláveis	Java, microkernels, Multithreading, OS distribuídos, www	IBM SP2, SGI Origin 2000, Digital TruCluster

Como pode ser visto na Tabela 1, em termos de tecnologias de hardware, a primeira geração criada foi a das válvulas. A segunda geração foi a dos transistores. A terceira geração foi marcada pelos circuitos integrados como o *SSI* (*Small-Scale Integrated*). Já na quarta geração, surgiram os microprocessadores de larga escala, ou circuitos *VLSI* (*Very Large-Scale Integrated*). E por último, veio a geração atual caracterizada pelos computadores paralelos e circuitos *ULSI* (*Ultra Large-Scale Integrated*).

Nas tecnologias de software, a primeira geração foi marcada pelo uso de linguagens de máquina e *assembly*. A segunda geração foi marcada pelas linguagens de alto nível como *Algol* e *Fortran*. Na terceira geração, predominou o uso da linguagem *C,* com sistemas multiprogramados e com *time-sharing*. Já na quarta geração, a programação paralela se firmava com os compiladores paralelos e bibliotecas para troca de mensagens. E na geração atual, as tecnologias de software existentes estão baseadas no suporte à programação orientada a objetos, e à programação paralela e distribuída.

Desde os primeiros trabalhos de *Von Neumann*, que as **arquiteturas paralelas** vêm sendo apresentadas como uma forma de obter uma maior capacidade de processamento para a execução de tarefas complexas. Esse aumento de processamento só podia ser obtido através de processadores mais velozes ou através do aumento do número de processadores empregados em conjunto. Entre os anos 50 e 60, as tecnologias de fabricação de máquinas **monoprocessadas**[3] eram suficientes para atender à demanda de processamento daquela época. Entretanto, a partir dos anos 70, as necessidades crescentes não eram mais suportadas por tais arquiteturas, tornando-se necessário a utilização de **técnicas de concorrência**[4] para alcançar o desempenho requerido.

Como o aumento da velocidade dos processadores esbarrava no custo e no limite da capacidade tecnológica para o desenvolvimento de circuitos mais rápidos, a tendência para o emprego de vários processadores trabalhando em conjunto foi a solução encontrada. Dessa forma, obteve-se uma maior capacidade de processamento, e definiu-se o termo **processamento paralelo,** o qual designa o uso de diferentes técnicas de concorrência para suportar as exigências de desempenho [3].

O desenvolvimento da área de processamento paralelo levou ao surgimento de máquinas especializadas em realizar grandes quantidades de operações por segundo, conhecidas como **supercomputadores**. Essas máquinas proporcionaram um maior desempenho na resolução de problemas, algo que encorajou os usuários a tentar solucionar problemas cada vez mais complexos [2].

Áreas em particular como, modelagem numérica e simulação, são consideradas extensões da ciência e engenharia, e que exigem alto desempenho. Além dessas áreas, diversas outras apresentam problemas que necessitam de supercomputadores para serem solucionados.

[3] **Máquinas monoprocessadas:** arquitetura de computador formada por um único processador (máquinas convencionais).
[4] **Técnicas de concorrência:** técnicas usadas para a sobreposição computacional, visando obter um melhor desempenho. As principais formas de concorrência são: temporal e espacial. Na primeira existe uma sobreposição das execuções no tempo, criando um ganho no desempenho final do processamento. Enquanto que na segunda, ocorre a sobreposição de recursos através da utilização de vários processadores, ou elementos de processamento que trabalham em paralelo na execução das tarefas que compõem o processamento. O ganho de desempenho ocorre devido a quantidade de unidades de processamento que trabalham em conjunto [3].

Alguns dos mais importantes grupos de problemas são os chamados **grandes desafios,** e não são capazes de serem solucionados em tempo hábil pelos computadores seqüenciais atuais [4]. Um problema típico e considerado um grande desafio é a previsão do tempo, onde cálculos complexos são necessários para a simulação de modelos atmosféricos. Conseqüentemente, para a obtenção de resultados precisos e rápidos, computadores extremamente poderosos são necessários. Outros exemplos são: dinâmica de fluídos, cálculos astronômicos e aplicações de banco de dados, como sistemas de informações gerenciais *(SIG),* os quais também necessitam além de alto desempenho, grande quantidade de memória e armazenamento em disco.

2.2 COMPUTAÇÃO PARALELA

O grupo chamado "grandes desafios", é exemplo de problemas que necessitam de computadores de alto desempenho para serem solucionados. Uma saída eficaz dá-se através do uso de **computadores paralelos**. Um computador paralelo consiste, basicamente, em um ou mais processadores, com um ou mais núcleos de processamento, que realizam operações computacionais simultâneas [2]. Com o uso de tal arquitetura, um problema pode ser teoricamente dividido em *n* subproblemas (onde *n* é o número de processadores disponíveis), sendo cada parte solucionada por um processador. O tempo ideal para o término desse processamento deve ser *t*/*n*, onde *t* é o tempo de processamento realizado em um computador seqüencial.

2.3 EVOLUÇÃO DAS ARQUITETURAS PARALELAS

A **computação paralela** não é uma área nova. Em 1920, *Vannevar Bush* apresentava um computador analógico capaz de resolver equações diferenciais em paralelo. O próprio *Von Neumann*, em seus artigos por volta de 1940, sugere, para resolver equações diferenciais, uma grade em que os pontos são atualizados em paralelo [2]. Entretanto, uma das primeiras máquinas a empregar o paralelismo do tipo sobreposição (*overlap*) foi o *UNIVAC I*. Essa máquina suportava a sobreposição de execuções de programas com as operações de entrada e saída (E/S) [3].

A velocidade dos computadores dependia da velocidade de execução das instruções, e da velocidade de transferência dos dados entre a memória e a CPU. Em 1960, a técnica de acelerar a busca de instruções na memória foi empregada na máquina *IBM 7094*, onde cada acesso à memória para busca de uma instrução já trazia a próxima instrução junta. Dessa forma, era eliminada metade dos acessos à memória para busca de instruções.

Posteriormente, surgia o entrelaçamento de memória como uma técnica que permitia o acesso simultâneo à memória pela divisão desta em diversos bancos. Cada um destes bancos podia ser acessado separadamente através de canais, pelos quais os dados podiam ser transferidos simultaneamente.

O emprego da técnica de ***pipeline***[5] aparece nos computadores *STRETCH* e *LARC*. O primeiro dividindo a execução da instrução em dois estágios: uma fase de busca e decodificação da instrução, e uma fase de execução da operação; enquanto no segundo, a execução das instruções foi dividida em quatro estágios: busca de instrução, operação de endereçamento e índice, busca do operando e execução da instrução [5].

Uma forma de melhorar a velocidade de transferência dos dados entre a memória e os processadores, foi o emprego de pequenas memórias rápidas, servindo como uma área de armazenamento temporário. Essas memórias rápidas foram chamadas de memórias ***cache***, e permitiam que as CPUs acessassem os dados nas memórias com mais velocidade. Seu funcionamento se baseia no princípio de que normalmente o acesso a dados e instruções na memória ocorre na forma seqüencial ou em endereços próximos.

Na evolução das arquiteturas, um dos problemas que apareceram e que dificultava o ganho de velocidade, foi a dependência entre instruções. Uma instrução, num determinado estágio do *pipeline*, pode depender do resultado de uma instrução prévia que ainda não completou seu processamento, e que, portanto, impedirá a instrução dependente de passar para o próximo estágio. Nas primeiras máquinas essa situação simplesmente causava um atraso na execução do *pipeline* de instruções. Para minimizar esse problema, surgiram computadores como o *CDC 6000,* que minimizavam o número de possíveis dependências através do emprego de um formato de instruções simples, além de hardware específico para funções separadas. Isto levou ao projeto de uma arquitetura com varias unidades funcionais, tais como somadores e multiplicadores, que executando em paralelo permitia um ganho na velocidade do processamento de instruções [2, 5].

Posteriormente, surgiram máquinas para a resolução de tarefas que necessitavam de uma grande quantidade de processamento repetitivo sobre um conjunto de elementos individuais de dados (os vetores de dados). Essas máquinas foram criadas para executar diretamente uma nova classe de instruções, as instruções vetoriais. Dessa forma são eliminadas as tarefas de controle do laço de instruções, simplificando a forma de funcionar dos *pipelines*. Esses computadores ficaram conhecidos como **máquinas vetoriais** ou **processadores vetoriais**, cujos primeiros exemplos foram o *IBM 2938*, o *CDC STAR-100* e o *TI-ASC* [2].

[5] **Pipeline:** o princípio da técnica de *pipeline* baseia-se no conceito de paralelismo temporal, onde a execução de uma nova tarefa pode ser iniciada antes que o resultado da tarefa anterior tenha sido gerado, desde que existam recursos disponíveis no processador para tal (diversos estágios no *pipeline*).

As técnicas de *pipeline*, de memória *cache* e de memória entrelaçada, são atualmente empregadas de forma usual em computadores seqüenciais ***SISD***. No entanto, possuem limitações de hardware, devido à tecnologia e ao software inerente ao tipo de programa utilizado (conjunto de instruções sendo executadas). Uma alternativa a essa situação é o emprego de diversas CPUs com suas respectivas memórias interconectadas, dando origem às máquinas com vários processadores, ou máquinas multiprocessadas.

Os computadores que empregam diversos processadores (***MIMD***), ou elementos de processamento (***SIMD***) para aumentar seu poder computacional, têm no *ILLIAC IV*, seu marco inicial [6]. Com seu projeto iniciado nos anos 60 e tornado operacional em 1972, o *ILLIAC IV*, que possuía uma arquitetura *SIMD*, foi considerado o primeiro supercomputador.

Nos anos 80, as máquinas conhecidas como supercomputadores eram o *CRAY*, o *FUJITSU* e outras que agregavam, além da capacidade vetorial, técnicas de sobreposição, permitindo que várias operações em paralelos fossem desenvolvidas em *pipelines* diferentes. O *CRAY-1*, por exemplo, além de trabalhar com registradores vetoriais ao invés de diretamente com a memória (o que já acelerava tempos de busca de instruções), possuía a capacidade de encadear operações vetoriais, minimizando os tempos de espera pelo término de uma operação para iniciar uma próxima. Esses computadores evoluíram para o emprego de várias unidades de processamento em conjunto, gerando as máquinas multiprocessadas, como o *CRAY XMP* [2, 5].

Por último, chegam ao mercado nos anos 90 as máquinas compostas por agregados de computadores, denominadas ***clusters*** computacionais. Primeiramente, eles eram formados de estações de trabalho (*COW – Cluster of Workstations*), tais como o *IBM SP2*. Mas logo se transformaram em *clusters* de *PCs*, nos quais máquinas comuns são interligadas através de redes de interconexão de alta velocidade, formando um conjunto de alto desempenho [5].

2.4 TAXONOMIA DOS COMPUTADORES PARALELOS

Os computadores paralelos podem ser classificados por diversos aspectos segundo sua arquitetura. Nessa seção, são apresentados três diferentes esquemas de classificação. O primeiro esquema classifica os computadores paralelos através de sua organização entre processador e memória. O segundo esquema, conhecido como classificação de *Flynn*, organiza os computadores através da combinação do fluxo de instruções e fluxo de dados. E o último esquema classifica os computadores em cinco arquiteturas paralelas diferentes, baseado nos principais modelos físicos existentes [2].

2.4.1 ORGANIZAÇÃO ENTRE PROCESSADOR E MEMÓRIA

Em termos de organização entre processador e memória como um critério de classificação de máquinas paralelas, três grupos principais podem ser identificados: **arquitetura de memória compartilhada**, **arquitetura de memória distribuída** e **arquitetura de memória compartilhada e distribuída**.

2.4.1.1 ARQUITETURA DE MEMÓRIA COMPARTILHADA

A principal característica da arquitetura de memória compartilhada (*Shared Memory*), é que todos os processadores (P) têm acesso à mesma memória, existindo apenas um único espaço de endereçamento global (Figura 1). Em tal arquitetura, a comunicação e a sincronização entre processos são feitas implicitamente através de variáveis compartilhadas.

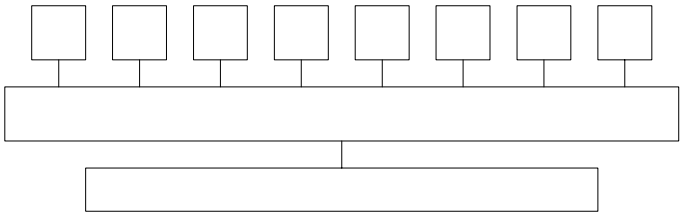


Figura 1: Arquitetura de memória compartilhada

Nessa arquitetura, os processadores são conectados aos módulos de memória através de algum tipo de interconexão. Esse tipo de computador paralelo é também chamado de *UMA (uniform memory access) ou* **multiprocessado**, permitindo acesso uniforme à memória, uma vez que todos os processadores a acessam com mesma latência e largura de banda.

Uma grande vantagem desse tipo de computador, é que devido ao compartilhamento de memória, a programação é mais conveniente, visto que os dados ficam disponíveis para todos os processos, não sendo necessário uma cópia destes. Outra vantagem, é que o programador não precisa se preocupar com questões de sincronização, uma vez que isso é realizado pelo sistema automaticamente (o que deixa o hardware mais complexo e conseqüentemente com um custo mais elevado). Entretanto, é difícil obter altos níveis de paralelismo com máquinas paralelas de memória compartilhada (a maioria dos sistemas não permite mais que 64 processadores), pois uma vez que a máquina já esteja construída, é muito difícil agregar mais processadores, devido ao hardware dedicado.

P P P

Rede

2.4.1.2 ARQUITETURA DE MEMÓRIA DISTRIBUÍDA

No caso dos computadores de memória distribuída (*Distributed Memory*), também conhecidos na literatura como **multicomputadores** [2], cada processador (P) possui sua própria memória privada (M), não existindo um espaço de endereçamento comum. A comunicação e a sincronização entre os processadores são feitas por troca de mensagens, através de uma rede de interconexão.

A Figura 2 mostra a organização entre os processadores (P) e os módulos de memória (M) em computadores de memória distribuída. Ao contrário dos computadores de memória compartilhada, que não são muito escaláveis, os computadores de memória distribuída conseguem bons níveis de escalabilidade, visto que não apresentam problemas de conflito no acesso a memória. Através do uso desse tipo de arquitetura, os computadores paralelos maciços (*MPPs*) podem ser construídos com centenas ou milhares de processadores.

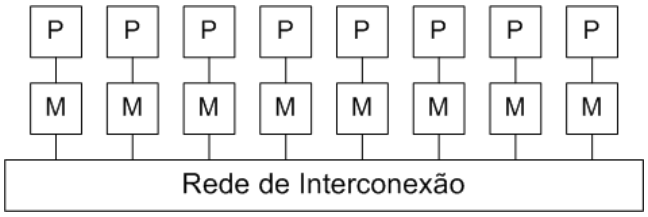


Figura 2: Arquitetura de memória distribuída

Uma representação típica dessa arquitetura são os *clusters* computacionais, os quais se tornam mais importantes a cada dia [7]. Em um *cluster*, cada nodo é um computador e a interconexão desses nodos dá-se com redes de baixo custo, como *Ethernet* e *Myrinet*. A maior vantagem dos *clusters* sobre as máquinas *MPPs*, é que os *clusters* apresentam um custo-benefício muito melhor. Entretanto, ainda não conseguem os desempenhos conseguidos pelas máquinas *MPPs* [5].

2.4.1.3 ARQUITETURA DE MEMÓRIA COMPARTILHADA E DISTRIBUÍDA

Para combinar as vantagens das outras duas arquiteturas citadas acima (fácil programação e alto nível de escalabilidade), uma terceira categoria de computadores foi estabelecida: computadores de memória compartilhada e distribuída (*Distributed Shared Memory - DSM*).

Nessa categoria, cada processador (P) tem sua própria memória local (M), conforme pode ser observado na Figura 3. Entretanto, ao contrário das máquinas de memória distribuída, todos os módulos de memória disponíveis formam um único espaço de

endereçamento, onde cada célula de memória mantém um sistema de endereçamento único. Para evitar o baixo nível de escalabilidade dos sistemas de memória compartilhada, cada processador usa um *cache* de memória, o que sustenta os possíveis conflitos existentes, e também as possíveis latências de interconexão. Porém, o uso de *cache* introduz uma série de problemas, como, por exemplo, manter os dados atualizados tanto na memória quanto no *cache*. Para solucionar esse problema, utilizam-se técnicas de consistência e coerência de *cache*, conforme podem ser verificadas em [8].

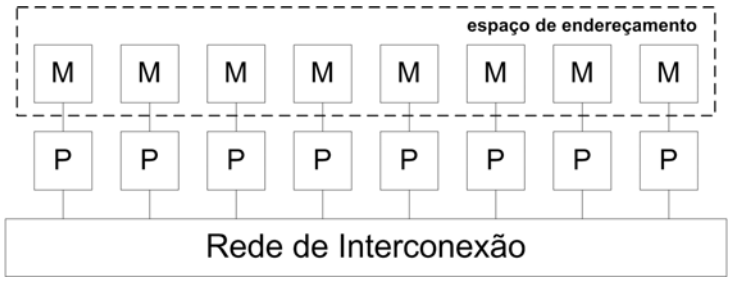


Figura 3: Arquitetura de memória compartilhada e distribuída

Uma alternativa de implementação para essa coerência de *cache*, chama-se Memória Virtual Compartilhada (*Shared Virtual Memory - SVM*). Nela o mecanismo de gerenciamento de memória de um sistema operacional tradicional é modificado para suportar esses serviços em nível de páginas ou de segmentos. A vantagem dessa abordagem é que não são necessárias alterações nas aplicações. Como exemplo, temos o projeto *SHRIMP* [9].

Outra possibilidade é não alterar o sistema operacional e utilizar bibliotecas de funções. Com essas bibliotecas, é possível converter o código desenvolvido para um espaço de endereçamento único em um código que suporte múltiplos espaços de endereçamento. Nesse caso, o código original tem que ser modificado para incluir compartilhamento de dados, sincronização e primitivas de coerência. O projeto *TreadMarks* [9] utiliza esse tipo de abordagem.

2.4.2 ESQUEMA DE CLASSIFICAÇÃO SEGUNDO FLYNN

Uma outra classificação para as máquinas paralelas foi proposta por *Flynn* em 1972 [10]. Nessa classificação os computadores são organizados segundo a execução de uma seqüência de instruções sobre uma seqüência de dados, sendo possível definir 4 categorias conforme a Tabela 2:

Tabela 2: Classificação das máquinas paralelas segundo Flynn		
Instrução/Dados	**Fluxo único de dados (SD)**	**Fluxo múltiplo de dados (MD)**
Fluxo único de instrução (SI)	***SISD***	***SIMD***
Fluxo múltiplo de instrução (MI)	***MISD***	***MIMD***

A classe das máquinas *SISD* (fluxo único de instrução e fluxo único de dados) refere-se às máquinas tradicionais de fluxo seqüencial, ou chamados computadores seqüências, os quais são baseados na arquitetura de *Von Neumann*. Conforme a Figura 4, o fluxo de instruções (linha contínua) alimenta uma unidade de controle (C) que ativa a unidade central de processamento (P). A unidade P, por sua vez, atua sobre um único fluxo de dados (linha tracejada), que é lido, processado, e reescrito na memória (M). Nessa classe de computadores, estão incluídas as máquinas monoprocessadas, como: microcomputadores pessoais e estações de trabalho.

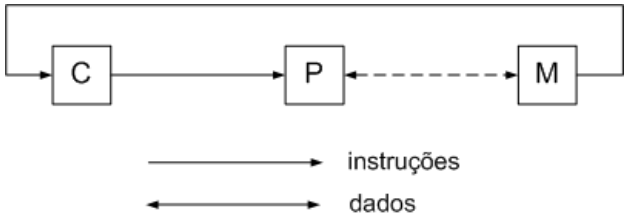


Figura 4: Ilustração da classe *SISD*

A classe *MISD* (fluxo múltiplo de instrução e fluxo único de dados) é uma classe que nunca foi implementada [2]. Nessa classe, múltiplos fluxos de instruções atuariam sobre um único fluxo de dados. Conforme a Figura 5, múltiplas unidades de processamento (P), cada uma com sua unidade de controle própria C, recebem um fluxo diferente de instruções. Essas unidades de processamento executam suas diferentes instruções sobre o mesmo fluxo de dados. Na prática, diferentes instruções operariam a mesma posição de memória simultaneamente, executando instruções diferentes, o que até os dias de hoje é tecnicamente impossível [5].

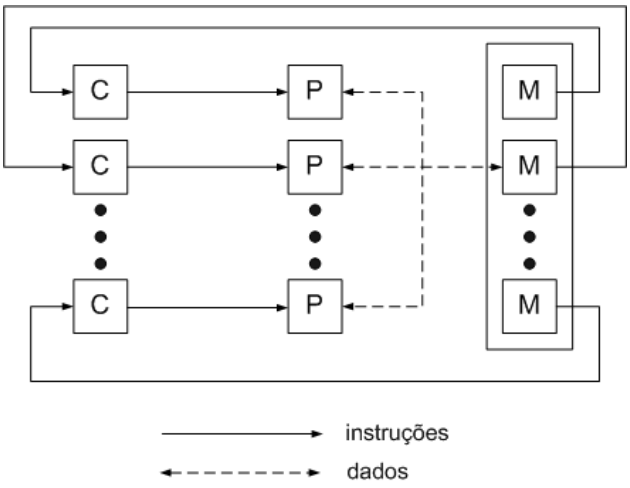


Figura 5: Ilustração da classe *MISD*

Os computadores paralelos concentram-se nas duas classes restantes: *SIMD* e *MIMD*. As representações mais importantes da classe *SIMD* (fluxo único de instrução e fluxo múltiplo de dados) são os computadores e processadores vetoriais. Essas máquinas têm apenas uma unidade de controle e instrumentação de memória, a qual controla múltiplos processadores. Tais processadores não foram projetados para uso geral, e sim para aplicações específicas como processamento de imagens. Conforme a Figura 6, uma única instrução é executada ao mesmo tempo sobre múltiplos dados. O processamento é controlado por uma única unidade de controle (C), alimentado por um único fluxo de instruções. A mesma instrução é enviada para os diversos processadores (P) envolvidos na execução, e todos os processadores executam suas instruções em paralelo de forma síncrona sobre diferentes fluxos de dados. Na prática, pode-se dizer que o mesmo programa está sendo executado sobre diferentes dados, o que faz com que o princípio de execução *SIMD*, assemelhe-se ao paradigma de execução seqüencial [2, 5]. É importante ressaltar que, para que o processamento das diferentes posições de memória possa ocorrer em paralelo, a unidade de memória (M) não pode ser implementada como um único módulo de memória, o que limitaria a somente uma operação por vez.

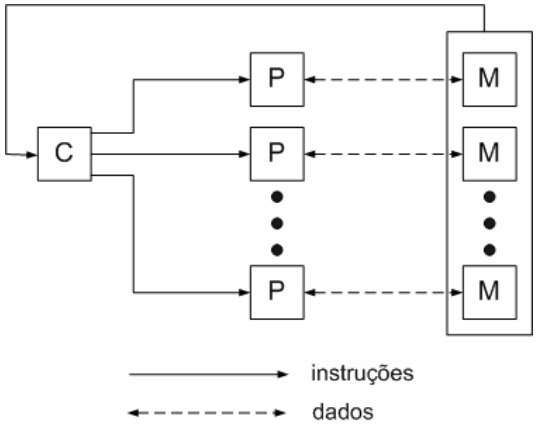


Figura 6: Ilustração da classe *SIMD*

A classe *MIMD* (fluxo múltiplo de instrução e fluxo múltiplo de dados) é a classe a qual pertence a maioria dos computadores paralelos atuais. Enquanto em uma classe *SIMD*, apenas um fluxo de instruções, ou seja, um único programa pode ser executado, em uma máquina *MIMD*, cada unidade de controle (C) recebe um fluxo de instruções próprio, conforme pode ser visto na Figura 7. Dessa forma, cada processador executa o seu próprio programa sobre seus próprios dados de forma assíncrona. Assim como na classe *SIMD*, a unidade de memória (M) não pode ser implementada como um único módulo de memória, o que permitiria apenas uma operação por vez. Na classe *MIMD* se enquadram servidores com múltiplos processadores, as redes de estações de trabalho e máquinas como *CM-5*, *nCube*, *Intel Paragon* e *Cray T3D*.

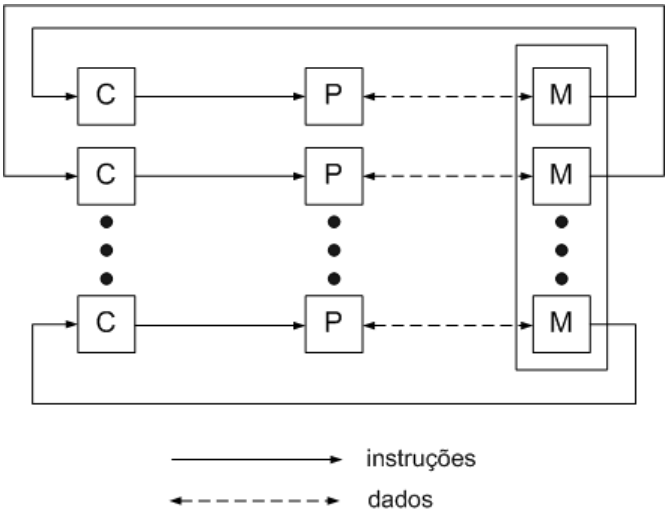


Figura 7: Ilustração da classe *MIMD*

2.4.3 CLASSES DE ARQUITETURAS PARALELAS

Segundo os autores [2, 4, 5, 8], é possível identificar as seguintes classes de arquiteturas paralelas, conforme a Tabela 3:

Tabela 3: Classes de arquiteturas paralelas

Arquiteturas de memória compartilhada	• **PVP** (Parallel Vector Processors) • **SMP** (Symetric Multiprocessors)
Arquiteturas de memória distribuída	• **MPP** (Massive Parallel Processors) • **DSM** (Distributed Shared-Memory Machine) • **COW** (Cluster of Workstation)

Atualmente, a maioria das máquinas paralelas está sendo comercialmente produzida, exceto aquelas baseadas na arquitetura *PVP*, as quais possuem diversos componentes customizados. As máquinas *PVPs*, ou Processadores Vetoriais Paralelos (*Parallel Vector Processors*), são sistemas compostos por poucos e poderosos processadores vetoriais, que são especialmente desenvolvidos para esse tipo de máquina. A comunicação entre os processadores é feita através de memória compartilhada, formando um espaço de endereçamento único que engloba todos os módulos de memória. Exemplos dessa arquitetura são: *Cray C-90*, *Cray T-90* e *NEC SX-4*.

Em contraste com as máquinas *PVPs*, as *SMPs*, ou Máquinas de Multiprocessadores Simétricos (*Symmetric Multiprocessors Machines*), possuem microprocessadores com *caches* on-*board* e off-*board*, e acesso à memória compartilhada através de barramentos de alta velocidade. Estes sistemas são considerados simétricos, porque todos os processadores possuem acessos semelhantes tanto à memória compartilhada quanto aos dispositivos de E/S e também aos serviços oferecidos pelo sistema operacional. Um problema dessa arquitetura é a falta de escalabilidade, que é limitada principalmente pelo uso centralizado de memória compartilhada, e também devido aos barramentos de interconexão, o que dificulta expansões de hardware, uma vez que a máquina já esteja construída. Como exemplos: *IBM R50*, *SGI Power Challenge* e *DEC Alpha server 8400*.

Os últimos três grupos, *MPPs*, *DSMs* e *COWs*, são máquinas de memória distribuída, o que permite vantagens de escalabilidade perante as limitações de arquiteturas de memória compartilhada, principalmente em aplicações que exigem um alto grau de paralelismo.

As máquinas maciçamente paralelas (*Massively Parallel Processors – MPPs*), geralmente referem-se a sistemas de computação de larga escala. Esses sistemas possuem diversos nodos individuais de processamento, compostos de microprocessadores e memória local, interconectados com outros nodos através de redes de alta velocidade e com baixa

latência de comunicação, o que pode ser escalável a centenas ou milhares de processadores. Os programas para essas máquinas consistem em múltiplos processos, cada um com seu próprio espaço de endereçamento de memória, e se comunicando através de troca de mensagens. A sincronização entre processos dá-se por operações de troca de mensagens, ao invés de operações sincronizantes com variáveis compartilhadas. As máquinas *Intel Paragon*, *Connection Machine CM-5* e a *IBM SP2*, são exemplos dessa arquitetura.

As máquinas *DSMs* (*Distributed Shared Memory*), ou máquinas com memória compartilhada e distribuída, provêem extensões de software e hardware que permitem um único espaço de endereçamento de memória, enquanto a memória física fica distribuída entre os nodos do sistema. O que diferencia essas máquinas das arquiteturas *MPP*, é a possibilidade de acesso às memórias remotas e a utilização de coerência de cache. Como exemplos dessa arquitetura, temos: *Stanford DASH* e *CRAY T3D*.

O último grupo de máquinas de memória distribuída (*COW – Cluster of Workstations*), também conhecida como máquinas agregadas [5], são baseadas no conceito de *cluster*, como os sistemas implementados em *Digital´s TruClusters*, *IBM´s SP2* e *Berkeley NOW*. De certo modo, os *clusters* são versões de custo mais baixo das máquinas *MPPs*, o que se torna uma alternativa interessante para instituições de pesquisa [11, 12]. Uma característica importante desse grupo, é que cada nodo é uma estação de trabalho completa (provavelmente sem alguns periféricos como: teclado, mouse e monitor), e que são conectados a outros nodos através de redes também de baixo custo (*Ethernet*, *FDDI*, *ATM* ou *Myrinet*). Em contraste com máquinas *MPPs*, as interfaces de rede são fracamente acopladas ao barramento de E/S, e cada nodo do *cluster* contém seu disco local e um sistema operacional completo, enquanto as *MPPs* não possuem disco local e apenas *microkernels* integrados.

Outro conceito importante e que possibilita estender a idéia de sistemas de larga escala através de conexões de máquinas distribuídas, são: "*Metacomputers*" [13], "*MetaSystems*" [15] e "*Computing Grids*" [14] ou "*Information Power Grid*" [14]. Um exemplo é o projeto multi-institucional conhecido como "*Globus Project*", que busca alto desempenho através de *grids* computacionais. Tais sistemas são compostos por arquiteturas de hardware fracamente acopladas, visando solucionar problemas particulares, e são considerados os sistemas mais poderosos já existentes [16].

2.5 PROGRAMAÇÃO PARALELA

Construir máquinas poderosas como as apresentadas nas seções anteriores, é apenas uma parte da história dos sistemas de computação paralelos. Tais supercomputadores devem ser programados, e softwares dedicados devem ser criados para que essas máquinas obtenham o

desempenho desejado. Para isso, nessa seção, sobe-se um nível na abstração desses sistemas, e trata-se dos processos que são executados nos processadores dessas máquinas paralelas.

Desenvolver programas para qualquer sistema de computador é difícil e exige bastante tempo. Entretanto, essa tarefa de desenvolvimento torna-se ainda mais complicada quando se trata dos detalhes inerentes ao paralelismo [17]. Enquanto a programação seqüencial já é largamente aceita e praticada pela maioria das pessoas, a programação paralela ainda é vista, em alguns casos, como uma rara e exótica subárea da computação, interessante e intelectualmente desafiadora, mas pouco relevante para muitos programadores [3].

No paradigma seqüencial de programação, o programador possui uma visão simplificada da máquina como sendo um único processador que pode acessar certa quantidade de memória. Já o paradigma de troca de mensagens é uma busca da portabilidade para a programação paralela. Neste paradigma, várias instâncias do paradigma seqüencial trabalham juntas. Isto significa que o programador pode imaginar vários processadores, cada um com a sua própria memória, e escrever um programa para executar em cada processador. Entretanto, a programação paralela, por definição, requer a cooperação entre os processadores para resolver o problema, o que torna necessário a implementação de algum meio de comunicação.

2.5.1 PARADIGMA DA TROCA DE MENSAGENS

Uma maneira natural de se programar em máquinas de memória distribuída é através da troca de mensagens, assim como, nas máquinas de memória compartilhada, é natural o uso de variáveis compartilhadas. Entretanto, é possível também inverter ou mesclar esses paradigmas, onde se usa troca de mensagem em máquinas de memória compartilhada e variável compartilhada em máquinas de memória distribuída [3]. Ambas as abordagens são praticadas. Contudo, o uso de variáveis compartilhadas em máquinas de memória distribuída exige camadas adicionais de software para prover uma visão compartilhada da memória que se apresenta fisicamente distribuída, conforme foi apresentado na seção 2.3.

A principal característica do paradigma da troca de mensagens é que os processadores comunicam-se através do envio e recebimento de mensagens. Assim, nesse modelo não há o conceito de memória compartilhada ou de processadores que possam acessar a memória de outro processador diretamente.

O paradigma da troca de mensagens vem se tornando cada vez mais popular. Uma das razões é o grande número de plataformas que oferecem suporte à troca de mensagens. Programas escritos neste paradigma podem executar em multiprocessadores de memória distribuída ou compartilhada, redes de computadores e sistemas com um único processador.

Segundo [3], existem três maneiras de programar em um ambiente de troca de mensagens:

- Desenvolvendo uma linguagem de programação paralela.
- Modificando uma linguagem seqüencial existente para manipular a troca de mensagens.
- Usando uma linguagem seqüencial de alto nível já existente, juntamente com uma biblioteca que forneça rotinas de troca de mensagens.

Existem exemplos para as três abordagens citadas. A linguagem *OCCAM*, que foi criada para ser usada no *Transputer*, é um exemplo para a primeira abordagem. Para a segunda abordagem, diversas linguagens seqüenciais foram estendidas para prover funcionalidades à troca de mensagem. Como exemplo, é possível citar *CC+*, que é uma extensão da linguagem *C++*. E *Fortran M,* como extensão da linguagem *Fortran*. Nessa dissertação, os estudos estão concentrados na terceira abordagem, onde são utilizadas linguagens de alto nível como *Java* e *C*, apoiadas em bibliotecas que ofereçam suporte a comunicação por troca de mensagem.

Existem algumas bibliotecas para troca de mensagem disponíveis, sejam elas comerciais ou não, e de plataformas especificas ou independentes. Duas bibliotecas muito usadas e que são para plataformas independentes, são: *PVM* (*Parallel Virtual Machine*) [18, 19] e *MPI* (*Message Passing Interface*) [20]. Um exemplo de biblioteca comercial para plataforma específica é *NCube*. Nessa seção, o paradigma da troca de mensagem não será abordado em detalhes. Apenas uma descrição abstrata das funções mais usadas será apresentada. Mais detalhes pode ser conseguido em [18, 20].

2.5.1.1 ROTINAS PARA TROCA DE MENSAGENS

As funções básicas do paradigma de troca de mensagem são o envio e o recebimento (***send*** e ***recv***), e são usadas respectivamente pelos processos ao enviar e receber mensagens como forma de comunicação. Essas primitivas podem ser tanto **síncronas** como **assíncronas**, conforme a Tabela 4:

Tabela 4: Primitivas básicas de comunicação

Síncronas	• **ssend** (bufer, destino) • **recv** (bufer, origem)
Assíncronas	• **send** (bufer, destino) • **irecv** (bufer, origem)

Um conhecimento importante que se deve ter quando se trabalha com funções de troca de mensagens, é a diferença entre síncrona e assíncrona. As funções síncronas, ou bloqueantes (*ssend* e *recv*), só retornam após a transferência dos dados ter sido completada. Antes disso, os processos ficam bloqueados, conforme a Figura 8.

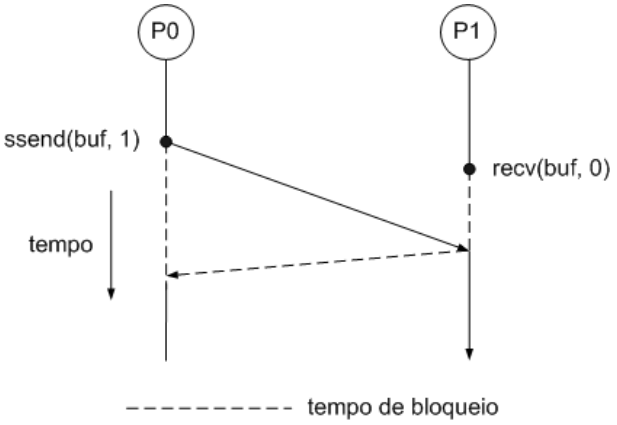


Figura 8: Troca de mensagem síncrona

Por outro lado, as funções assíncronas, ou não bloqueantes (*send* e *irecv*), retornam imediatamente após a comunicação ter sido iniciada. Nesse caso, a mensagem a ser enviada deve ser copiada para um *buffer*, e logo após é transferida pelo sistema para o *buffer* do processo destino. A função de recebimento deve ler essa mensagem do *buffer* tão logo ela esteja disponível.

Esse comportamento não bloqueante não apresenta problemas para a função *send*, mas requer alguns cuidados para a função *irecv*, pois uma vez que a função de recebimento não bloqueante retorna imediatamente, nada garante que a mensagem foi entregue ou não. Para esse propósito, existe uma outra função importante e que pode ser usada para verificar se uma mensagem a ser enviada está disponível ou não, que é a:

- **test**(origem)

Através dessa função esse problema pode ser solucionado, retornando *true* caso a mensagem esteja disponível, ou *false* caso contrário. Essa função de teste geralmente é usada em um laço para o recebimento, onde a função *irecv* só é chamada quando a função de teste retornar *true*, indicando que a mensagem está disponível no seu *buffer*.

A maioria das bibliotecas de troca de mensagem oferece bem mais funções do que as apresentadas acima. Por exemplo, é possível especificar grupos de processos para enviar mensagens de maneira coletiva como: *gather*, *scatter*, *reduce*, entre outras. Mais detalhes e a

lista completa de funções de comunicação podem ser conseguidos em [18, 19, 20]. Entretanto, para esse trabalho, as funções apresentadas são suficientes.

2.5.2 PARADIGMA DA VARIÁVEL COMPARTILHADA

A principal diferença desse paradigma para o paradigma da troca de mensagens é o uso de variáveis compartilhadas, ao invés de troca de mensagens. Como já foi visto na Seção 2.2, em ambientes de memória compartilhada, todos os processos têm acesso a uma memória global, com um único espaço de endereçamento. Conseqüentemente, qualquer local da memória pode ser acessado por qualquer processo. Isso requer mecanismos de sincronização para evitar que uma mesma variável seja alterada ao mesmo tempo por mais de um processo, em um mecanismo chamado de exclusão mútua.

Como nesse trabalho o paradigma utilizado é o baseado em troca de mensagens, mais detalhes sobre as variáveis compartilhadas podem ser conseguidos em [3, 4, 12].

Capítulo 3

DOMÍNIO DO PROBLEMA: TESTE E DEPURAÇÃO

Neste capítulo é apresentada uma revisão sobre teste de software e depuração. Inicialmente são apresentados alguns conceitos básicos relacionados ao ciclo de vida de um software, onde é dada ênfase a um modelo de desenvolvimento, assim como modelos de maturidade do processo de teste. Com isso, contextualizam-se as atividades de teste e depuração no ciclo de desenvolvimento de software. No decorrer do capítulo são feitas considerações gerais sobre teste, e uma revisão e classificação das principais nomenclaturas envolvidas. Em seguida são apresentadas às fases do teste e as principais técnicas e critérios de teste, onde é dada uma atenção especial para o teste de software funcional, dando destaque aos modelos de uso por constituírem o alvo do presente trabalho.

3.1 MODELO DE CICLO DE VIDA DE SOFTWARE

O processo de desenvolvimento profissional de sistemas de software é chamado de engenharia de software[6]. Esse processo cobra a aplicação e o uso de métodos e ferramentas na construção de sistemas de software.

Alguns dos principais aspectos da engenharia de software são: o gerenciamento e a coordenação de projetos, seus componentes e seus relacionamentos, assim como o entendimento do processo de desenvolvimento de sistemas. Basicamente todo esse conhecimento deve consistir em modelos de ciclo de vida, convenções de trabalho e garantia de qualidade de processo. Como conseqüência desses aspectos, um sistema de software deve incluir não somente programas de computador, mas também a documentação necessária para o seu desenvolvimento, operação e manutenção [21].

Modelos de ciclo de vida definem a ordem entre diferentes atividades e suas relações no processo de desenvolvimento de software. Basicamente três modelos podem ser

[6] Engenharia de Software: Disciplina que aplica técnicas da engenharia para produzir softwares de alta qualidade e baixo custo.

distinguidos [21]: o *waterfall-model* ou *linear-phases*, o *spiral-model* e o *cyclic-model*. Uma abordagem simples é representada pelo *waterfall-model*. Este modelo consiste em diversas fases cronológicas, conforme apresentado na Figura 9, onde cada fase é concluída com uma etapa de validação.

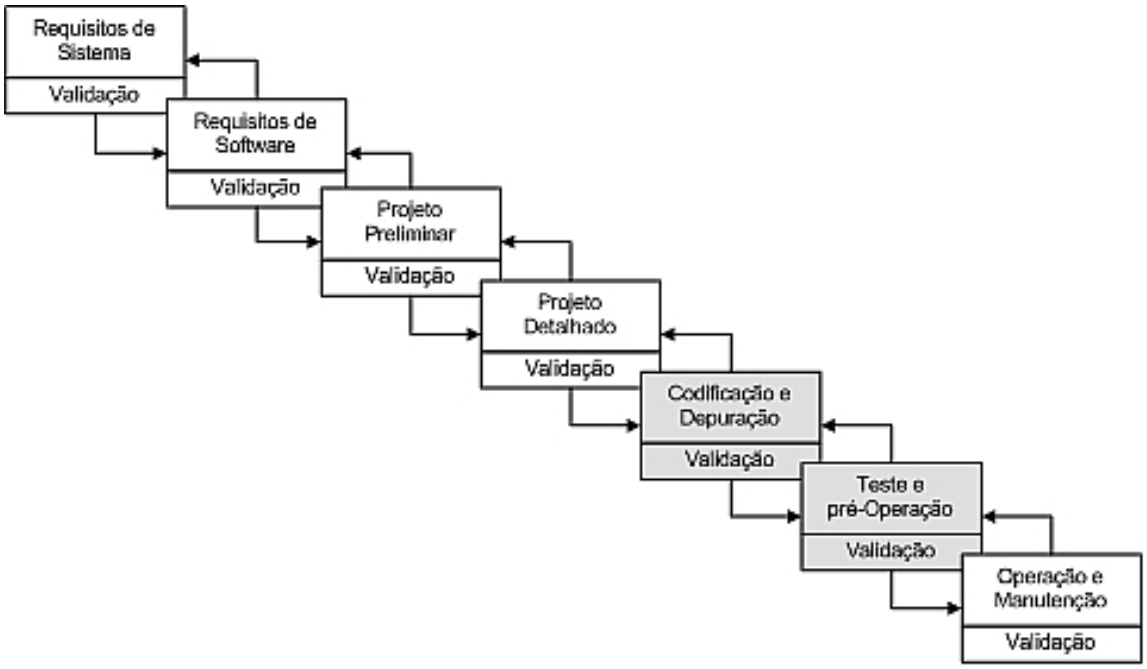


Figura 9: O modelo waterfall

Essas etapas de validação determinam o prosseguimento para as próximas fases do processo. Em casos de resultados insatisfatórios durante alguma dessas fases de validação, é necessário que o processo retorne à fase anterior. Entretanto, a cada avanço de fase é indispensável que os resultados sejam bem compreendidos e, de preferência, documentados. Se essas regras não forem seguidas, erros críticos podem ser detectados demasiadamente tarde, geralmente durante a fase de teste e depuração, e quanto mais tarde se der a correção desses erros, mais esforços serão necessários para a conclusão do projeto [22].

As fases de teste e depuração são os pontos de interesse dessa dissertação e serão abordadas em mais detalhes nas próximas seções.

3.1.1 A EVOLUÇÃO DA ENGENHARIA DE SOFTWARE

A busca por produtos de software de alta qualidade tem pressionado os profissionais ligados a área da engenharia de software a identificar e quantificar fatores de qualidade, como:

usabilidade, testabilidade e confiabilidade. Além desses fatores, outras práticas de desenvolvimento, também ligadas à engenharia, visam dar suporte a criação de produtos de qualidade, como: plano de projeto, gerenciamento de requisitos, desenvolvimento de especificações formais, reuso de código e projeto, inspeções e revisões, desenvolvimento e aplicação de ferramentas CASE, entre outras.

Como complemento à identificação dessas técnicas de desenvolvimento e melhores práticas individuais de gerenciamento da qualidade[7], pesquisadores ligados à engenharia de software concluíram que é importante integrar todos esses esforços dentro de um contexto relacionado a um processo de desenvolvimento de software de alta qualidade. Com isso, surgiram alguns modelos de maturidade e qualidade de processo, como *CMM* (*Capability Maturity Model*), *ISO-15504-SPICE* e *Bootstrap* [23, 24, 25, 26].

Durante anos esses modelos de melhoria de qualidade de processo vêm sendo aplicados e aperfeiçoados com o apoio tanto das indústrias quanto dos centros de pesquisas acadêmicas. Entretanto, esses modelos são modelos de alto nível e visam a melhoria do processo como um todo, não oferecendo um suporte adequado para a progresso de etapas intermediárias do processo, como o teste e a depuração. Principalmente na última década, quando os grupos de teste de software começaram a surgir, tornando-se o teste uma atividade independente do processo de desenvolvimento, embora sempre com etapas integradas [22].

A criação de um processo independente de teste demandou necessidades de melhorias metodológicas e métricas, visto que estas já existiam no processo de desenvolvimento de software original, mas que precisavam ser adaptadas a este novo processo. Em 1996 foi desenvolvido então, por técnicos do *Illinois Institute of Technology* (*Illinois State University*), o modelo *TMM* (*Test Maturity Model*). Esse modelo passou a permitir a realização de diagnósticos do processo de teste dentro das empresas, possibilitando medidas dos seus níveis de maturidade, e com isso definir um roteiro gradativo de melhorias.

Atualmente existem vários modelos de avaliação da maturidade do processo de teste. Alguns como o *TPI* (*Test Process Improvement*) que é bastante usado na Europa, e outros como o *TMM* (*Test Maturity Model*) e *TCMM* (*Test Capability Maturity Model*), que são mais populares nos Estados Unidos. Entretanto, já existe um grande esforço no sentido de integrar os modelos de maturidade de teste (*TMM*), com os modelos de maturidade da capacitação para software (*CMM*). Enquanto isso não ocorre, é preciso tratar o teste como um processo separado e com características próprias, o que justifica a existência de modelos como o *TMM*. Mais informações sobre modelos de maturidade e qualidade de software pode ser encontrado em [26].

[7] Os fatores de qualidade serão abordados na Seção 3.2.2.

3.2 TESTE DE SOFTWARE E DEPURAÇÃO

3.2.1 CARACTERÍSTICAS GERAIS

O processo de desenvolvimento de software é descrito como uma série de fases, procedimentos, e passos que resultam na criação de um produto de software. Conforme o escopo dessa dissertação, nosso interesse está focado principalmente nas fases de "teste e depuração", as quais são etapas responsáveis por "encontrar falhas no programa em teste" e "corrigir essas falhas encontradas", respectivamente.

A fase de teste é geralmente iniciada sempre que um módulo ou um programa completo tenha sido implementado, uma vez que seja necessária a revisão deste na busca por comportamentos incorretos. Por esta razão, a atividade de **teste** está fortemente relacionada com as atividades de **verificação** e **validação** (VV&T[8]). Enquanto a primeira é utilizada para provar o quanto um programa está correto de acordo com as suas especificações de comportamento, a segunda visa avaliar a execução do programa através de um conjunto limitado de entradas.

Mais especificamente, a verificação é um conjunto de atividades que garante que o software implementa corretamente as funções de comportamento especificadas. Nesse sentido, um programa é dito correto, quando este cumpre com as suas especificações, permitindo aos programadores terem certeza de que "construíram certo o produto".

Em contraste com a verificação, o processo de validação é aplicado para conferir se um sistema ou algum módulo em particular, correspondente a um determinado requisito, está de acordo com o resultado observado. Para tal processo, o programa deve ser executado em um ambiente de monitoração ou simulação. Nesse contexto, a etapa de teste pode também ser descrita como a aplicação interativa da validação, ajudando os programadores a ter certeza de que estão "construindo o produto certo".

3.2.2 TERMINOLOGIA E CONCEITOS BÁSICOS

Para uma melhor compreensão dos conceitos tratados nessa dissertação, nesta seção será definido um vocabulário relacionado aos processos de teste de software e depuração. O conhecimento desses termos básicos é essencial para assegurar que os conceitos apresentados estão baseados em um vocabulário comum, e que este é aceito tanto pela academia quanto pela indústria.

[8] VV&T(Verificação, Validação e Teste): Atividade relacionada com garantia de qualidade agregada ao processo de desenvolvimento de software.

Muitos dos conceitos a serem tratados estão baseados em termos descritos no *IEEE Standards Collection for Software Engineering.* Esses padrões incluem o *IEEE Standard Glossary of Software Engineering Terminology*, que é um dicionário dedicado à descrição de vocabulários referentes à engenharia de software [27], e de onde foram retiradas as seguintes definições:

- **Erro (*Error*)**[9]**:** um erro é um engano, uma ação humana que produz um resultado incorreto. Esse erro causa uma diferença entre o valor obtido com a execução do programa e o valor esperado na especificação. O erro pode ser cometido por programadores, analistas e testadores em situações adversas. Como exemplo, pode-se citar um documento com uma especificação errada, ou um código mal escrito por falta de entendimento de sua especificação.
- **Falha (*Fault*):** uma falha é a incapacidade de um software ou componente de software de realizar as funções esperadas com o desempenho exigido pelas suas especificações. Também é considerada como a produção de uma saída incorreta em relação as suas especificações.
- **Defeito (Failure):** um defeito é introduzido em um software como o resultado de um erro. É um passo, processo ou definição de dados incorretos. Algumas vezes os defeitos são considerados "*bugs*".
- **Casos de Teste (*Test Case*):** a abordagem básica para a detecção de falhas em um programa é selecionar um conjunto de dados de entrada (*input data*), e então executar esse programa com esses dados em um conjunto de condições específicas. A definição do resultado do teste só é possível, caso se tenha os resultados desejados (*output data*). Assim, torna-se possível comparar o que realmente aconteceu na execução do programa com o que era esperado. Um caso de teste deve conter todas essas informações, ou seja, além dos dados externos à aplicação que servem como um estímulo ao teste (dados de entrada), exercitando a aplicação em teste, este deve conter também parâmetros de execução, e os resultados esperados para as saídas da execução do programa.
- **Conjunto de Teste (*Test Set*):** o conjunto de teste é um grupo de casos de testes relacionados, ou um grupo de casos de teste e procedimentos de teste. Entretanto, um grupo de testes relacionados e que são executados em conjunto é conhecido como *test suite.*
- **Oráculo de Teste (*Test Oracle*):** um oráculo de teste é um documento ou um módulo (componente) de software, que permite determinar quando um teste é bem sucedido ou não, ou seja, quando um caso de teste encontrou falhas durante o teste ou não,

[9] No contexto dessa dissertação, os termos **defeito** e **erro** são tratados como **erro** (causa). E o termo **falha,** como sendo a conseqüência, ou a identificação, a manifestação de um comportamento incorreto do programa devido a algum tipo de **erro**.

respectivamente. O programa ou o documento que produz ou esclarece os resultados do teste, pode ser considerado um oráculo de teste.

- **Qualidade de Software (*Software Quality*):** no *IEEE Standard Glossary of Software Engineering Terminology* [27], há duas definições para o termo qualidade. "A primeira refere-se ao quanto um sistema, componente de sistema ou um processo cumpre com seus requisitos especificados". "E a segunda refere-se ao quanto um sistema, componente de sistema ou um processo cumpre com as necessidades e expectativas de usuários e clientes".

  Para determinar a qualidade de um sistema, são usados alguns atributos, os quais são características que inspiram qualidade. Como essas definições são esforços internacionais, muitos desses atributos não possuem uma tradução literal para o português. Dessa forma, nesse trabalho, os nomes serão apresentados em inglês, e apenas o significado será traduzido para o português.

  Alguns exemplos desses atributos são:

  - ***Correctness***: certeza de que um sistema executa as suas funções pretendidas.
  - ***Reliability***: certeza de que um software é capaz de realizar suas funções requeridas sobre certas condições pré-definidas, e por um período de tempo pré-definido.
  - ***Usability***: relaciona-se à quantidade de esforço necessário para aprender a operar, preparar entradas e interpretar as saídas do software.
  - ***Integrity***: relaciona-se à habilidade do sistema de suportar ataques intencionais e acidentais.
  - ***Portability***: relaciona-se a capacidade do software de ser transferido de um ambiente computacional para outro diferente.
  - ***Maintainability***: relaciona-se a capacidade de manutenção de software.
  - ***Interoperability***: relaciona-se ao esforço necessário para interligar ou acoplar um sistema a outro.
  - ***Testability***[10]: este atributo desperta mais interesse aos programadores e testadores, do que aos clientes, e pode ser definido de duas maneiras:
    - Relaciona-se com a quantidade de esforço necessária para testar um software, assegurando que este realiza suas funções de

[10] Os testadores devem trabalhar em conjunto com analistas, projetistas e desenvolvedores durante o ciclo de vida do software para assegurar que os requisitos de testabilidade são atingidos.

acordo com as suas especificações (em relação ao número de casos de teste necessário para que isso ocorra).

- Relaciona-se com a capacidade do software em revelar falhas sobre condições de teste (isso porque alguns softwares são projetados de tal forma que os erros ficam escondidos em situações de teste).

É importante lembrar que os termos **falha**, **erro** e **defeito,** podem ser encontrados na literatura da área de teste com outras interpretações, embora às apresentadas acima façam parte de um padrão mundial estabelecido. Entretanto, esses termos também podem ser utilizados em outras áreas não relacionadas ao teste de software, como a Tolerância a Falhas ou Dependabilidade.

Diferente da abordagem do teste de software, que busca encontrar as falhas existentes no software e corrigir essas falhas, a tolerância a falhas busca garantir a capacidade de um sistema em manter seu correto funcionamento mesmo na presença de falhas. Essa capacidade aumenta a confiabilidade desses sistemas, visto que as falhas são inevitáveis, mas as conseqüências destas podem ser evitadas [28].

A principal diferença, ou a principal confusão causada por esses conceitos, é que no teste, o termo **erro** é usado como causa de uma **falha**. Enquanto na tolerância a falhas, esses termos são tratados de maneira inversa, onde a **falha** é a causa dos **erros**, podendo esta ser considerada de hardware, software, humana ou intencional (falha maliciosa). E ainda ter uma causa, natureza, duração, extensão e valor [29].

Embora a tolerância a falhas seja uma área importante quando se trata de sistemas computacionais, uma revisão dos seus conceitos foge um pouco do escopo dessa dissertação. O importante mesmo é delimitar esses conceitos para a área de teste de software, que será tratada no resto do texto.

3.2.3 PROBLEMAS E DIFICULDADES DO TESTE

Um dos principais problemas do teste de software é que, de maneira geral, este pode no máximo demonstrar a existência de falhas, mais nunca provar a ausência destas [21]. Além disso, testes exaustivos são geralmente impossíveis de serem executados, pois o domínio funcional de uma aplicação em teste pode ser infinito, resultando em uma quantidade muito grande de casos de teste.

O custo financeiro das etapas de teste também é um fator que dificulta esse processo, visto que este pode chegar a custar mais de quarenta por cento do valor total de um sistema [21]. Outro fator importante, é que geralmente as primeiras versões de um programa quase

nunca estão corretas. Conseqüentemente é provável que qualquer programa contenha algum tipo de erro. Uma razão para isso é a própria complexidade envolvida no processo de desenvolvimento, algo existente tanto na interpretação de uma especificação, como no sincronismo entre as diversas equipes envolvidas.

O critério de parada também é outra razão que dificulta o teste. Basicamente existem duas possibilidades: uma estatística, onde estatísticas de erros indicam certo grau de confiabilidade; e outra sistemática, que busca determinar métodos para identificar propriedades funcionais, além de classes de possíveis erros.

3.3 O PROCESSO DE TESTE

A atividade de teste de software consiste basicamente em uma análise dinâmica do produto a ser testado, sendo considerada uma atividade relevante para a identificação de falhas e eliminação de erros existentes em um sistema. O conjunto de informações oriundas da atividade de teste é significativo para as atividades de depuração, manutenção e estimativa de confiabilidade de software.

O teste de software envolve basicamente quatro etapas que são executadas durante o ciclo de desenvolvimento do software, que são: planejamento do teste, projeto de casos de teste, execução do teste e coleta dos resultados, e avaliação dos resultados coletados. A concretização dessas etapas se dá em três fases: o **teste de unidade**, o **teste de integração** e o **teste de sistema**.

O **teste de unidade** concentra seus esforços na menor unidade de projeto, identificando falhas relacionadas a erros de lógica e de implementação em cada módulo individual de software. O **teste de integração** é uma atividade sistemática aplicada durante a integração da estrutura do programa, visando descobrir erros associados às interfaces entre os módulos, ou seja, constrói a estrutura do programa determinada pelo projeto a partir dos módulos testados no nível de unidade. E o **teste de sistema**, que deve ser realizado após o teste de integração, visando descobrir erros de funções e características de desempenho que não estejam de acordo com a especificação.

As etapas do teste, assim como as etapas da depuração, geralmente ocorrem em ciclos[11], onde o programa pode ser executado sucessivamente até que se obtenha uma confiabilidade desejada. O ciclo do teste é usado para detectar computações incorretas, onde o programa é testado com um conjunto de casos de teste que exercitam o seu comportamento. Enquanto o ciclo da depuração provê informações sobre os estados do programa e também sobre resultados intermediários durante a execução. A idéia dessas etapas é identificar as

[11] O ciclo do teste e da depuração será apresentado no Capítulo 4.

falhas, e localizar e corrigir erros do programa baseado na comparação dos resultados obtidos com os resultados esperados [30].

Uma questão importante da atividade de teste, independentemente da fase, é a avaliação da qualidade de um determinado conjunto de casos de teste, visto que é impraticável utilizar todo o domínio de dados de entrada para avaliar os aspectos funcionais e operacionais de um produto em teste. Assim, o objetivo do teste é utilizar casos de teste que tenham alta probabilidade de encontrar a maioria dos erros com um mínimo de tempo e esforço. Portanto, um teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe [21].

A aplicação do teste segue diversas técnicas que oferecem perspectivas diferentes, abordando diferentes classes de erros, e devendo ser utilizadas de forma complementar. No entanto, elas diferenciam-se umas das outras pela origem das informações utilizadas no processo de teste, ou seja, quais objetos de teste devem ser testados, e o que deve ser testado nesses objetos. Para a engenharia de software, essas informações representam os **critérios** e **requisitos** de teste, e são indispensáveis para a especificação de um caso de teste.

3.3.1 TÉCNICAS DE TESTE DE SOFTWARE

Existem diversas técnicas para geração de casos de teste, e entre as principais estão: a **funcional** *(Black Box*), a **estrutural** (*White Box* ou G*lass Box*) e a baseada em **métodos estatísticos**.

Na técnica **funcional**, os critérios e requisitos de teste são estabelecidos a partir de uma função de especificação do software, onde o objetivo é determinar se o programa satisfaz aos requisitos funcionais e não-funcionais que foram especificados. O termo B*lack Box* se deve ao fato da aplicação ser vista como uma caixa fechada, ou seja, uma caixa onde não se tem acesso ao seu conteúdo, ou a sua estrutura e comportamento interno (código do programa).

Na técnica **estrutural**, os critérios e requisitos de teste são derivados essencialmente a partir das características de uma particular implementação em teste, o que requer a inspeção do código fonte, e a seleção de casos de teste que exercitem partes do código e não de sua especificação. Por essa técnica permitir o acesso a sua estrutura interna (transparência do código – *Glass Box*), é que é chamada de caixa-branca (*White Box*).

Na técnica baseada em **métodos estatísticos**, os critérios e requisitos de teste permitem o uso de inferências estatísticas para computar aspectos probabilísticos do processo de teste (resultados do teste), tais como confiabilidade, tempo médio para a ocorrência de falha (*MTTF*) e tempo médio entre falhas (*MTBF*). Observa-se também o uso de métodos estatísticos no estabelecimento de critérios para geração de casos de teste baseados em

máquinas de estados finito (*FSM*), cadeias de *Markov* (*MC*) e mais recentemente baseados em Rede de Autômatos Estocásticos (SAN).

O teste estatístico não deve ser entendido como uma técnica de teste de software que venha a substituir o teste funcional ou estrutural. Na verdade, ele visa empregar princípios estatísticos e probabilísticos no processo de teste, utilizando-se das técnicas tradicionais abordadas (funcional e estrutural). Assim, as inovações do teste estatístico se apresentam mais em termos do processo de teste do que efetivamente em técnicas de teste [31].

Em [31], é apresentado o processo típico do teste estatístico baseado em **modelos de uso**. Este é dividido em algumas etapas, conforme a Figura 10:

- **Análise da especificação:** o modelo de uso deve ser desenvolvido partindo-se da especificação do comportamento correto do sistema. Este pode ser definido através de uma especificação formal, documentação dos requisitos, manual do usuário, protótipo, etc.
- **Desenvolvimento da estrutura do modelo:** são identificados os estados e os arcos de transição entre eles através de um processo manual.

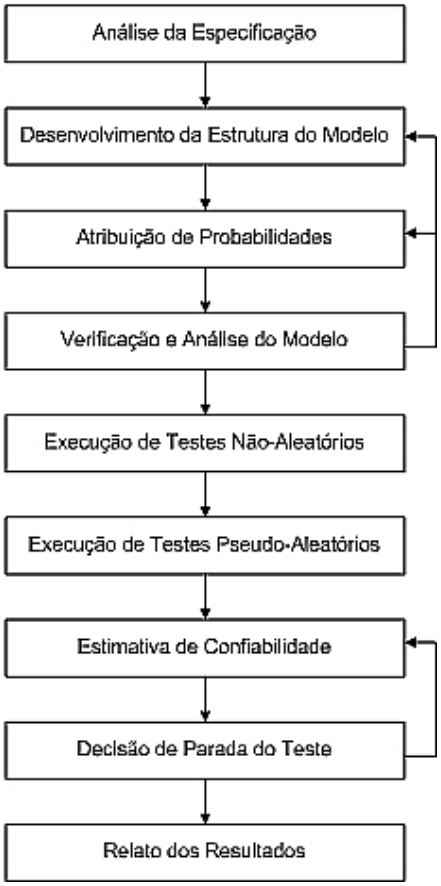


Figura 10: Etapas do teste estatístico

- **Atribuição de probabilidades:** as probabilidades de transição entre os estados do modelo são atribuídas manualmente ou calculadas automaticamente a partir da análise de uso.
- **Verificação e análise do modelo:** nesta etapa são realizados cálculos sobre o modelo, no intuito de apoiar o planejamento dos testes, a validação do modelo e assim por diante.
- **Execução de testes não aleatórios:** são gerados casos de teste para cobrir todos os arcos do modelo, seguindo a ordem das probabilidades de ocorrência.
- **Execução de testes pseudo-aleatórios (*random-testing*):** são gerados testes pseudo-aleatórios a partir do modelo, podendo ser executados de maneira automática ou manual.
- **Estimativa da confiabilidade**: é realizada a análise dos registros do teste aleatório no intuito de estimar a confiabilidade a partir do registro das falhas ocorridas e dos estados em que ocorreram.
- **Decisão de parada do teste:** consiste na avaliação do registro de teste, decidindo quando o teste deve parar ou prosseguir.
- **Relato dos resultados:** depois de encerrados os testes, seus resultados podem ser utilizados para decidir sobre a liberação de um produto, avaliar o grau de controle do processo de desenvolvimento, avaliar o desempenho de novos elementos integrados ao sistema e uma série de outros usos.

3.3.2 FORMALISMO PARA O TESTE ESTATÍSTICO

O teste estatístico com o passar dos anos passou a despertar grande interesse, pois viabiliza a superação de alguns pontos fracos de outras estratégias de teste [31]. Ele geralmente está baseado em **modelos de uso**, os quais podem descrever possíveis comportamentos de um determinado software.

A estrutura de um modelo de uso é composta por um conjunto de estados e transições entre esses estados, constituindo um grafo. Os nodos do grafo representam os estados da aplicação, e os arcos do grafo representam as transições entre os estados. Esta estrutura descreve os possíveis usos do software. Entretanto, ao associar uma distribuição de probabilidade à estrutura do modelo, este passa a descrever o uso esperado do software [31].

Devido ao modelo de uso constituir uma representação formal do software modelado, este pode ser aplicado a diversas fases do ciclo de vida do software. Ele pode ser utilizado para validar os requisitos do sistema, avaliar a complexidade de um sistema, apoiar o processo

de verificação do software, gerar automaticamente casos de teste, direcionar testes, identificar freqüência de eventos, projetar custos e recursos para o teste, definir critérios quantitativos do teste, critérios de parada, confiabilidade, entre outras aplicações [32].

No âmbito do teste estatístico, tais modelos permitem ao engenheiro de teste[12] visualizar caminhos críticos mais suscetíveis a falha, direcionando os esforços do teste nesse sentido. Essa vantagem se dá através da análise das probabilidades de ocorrência, associadas a cada uso do software.

Os modelos de uso geralmente são representados por algum tipo de formalismo. O primeiro formalismo utilizado no teste estatístico foi Cadeias de *Markov*. Entretanto, existem estudos recentes em torno de um outro formalismo chamado Rede de Autômatos Estocásticos (SAN).

3.3.3 CADEIAS DE MARKOV

Um processo estocástico é definido como um conjunto de variáveis aleatórias definidas em um espaço de probabilidades e indexadas por um parâmetro. Esse parâmetro geralmente refere-se a um conjunto de índices do tempo do processo, ou um intervalo de tempo. Se tivermos tempo discreto ($T = 1, 2, 3, ...$), temos um processo estocástico discreto. Se tivermos tempo contínuo ($T = 0 < t < +\infty$), temos um processo estocástico de tempo contínuo. Um processo estocástico é definido como um processo markoviano, quando não possuir memória em relação ao passado do sistema. Isso significa que apenas o estado atual do sistema influencia o próximo passo para a atingibilidade de estados futuros [33].

É possível representarmos o comportamento de um sistema descrevendo todos os diferentes estados que este venha a apresentar e indicando as transições possíveis de um estado para outro durante a sua execução. Este sistema pode ser representado como um processo markoviano, quando o tempo gasto em cada estado apresenta-se exponencialmente distribuído. A este processo markoviano está associado um conjunto de estados, sendo que este pode assumir apenas um estado em qualquer momento. A evolução do sistema é representada por transições do processo de um estado para outro, transições estas que, assume-se, ocorrem de maneira instantânea (sem consumir tempo).

Quando o espaço de estados de um processo markoviano é discreto (número finito de estados enumeráveis), o processo é chamado Cadeia de *Markov*. As cadeias de *Markov* são classificadas também em relação a uma escala de tempo, sendo cadeias de *Markov* de tempo discreto (DTMC) e cadeias de *Markov* de tempo contínuo (CTMC) [34]. Nas cadeias de tempo discreto (DTMC), temos probabilidades condicionais de ocorrem transições de um

[12] Engenheiro de Teste: pessoa responsável pela criação dos testes e gerência de sua execução.

estado para outro. Estas probabilidades de transição podem ser representadas como P_{ij} (probabilidade de transição do estado *i* para o estado *j*) nas cadeias homogêneas, cujas transições independem do tempo, ou como $P_{ij}(n)$ (probabilidade de transição do estado *i* para o estado *j* em um tempo *n*) nas cadeias não-homogêneas, cujas transições dependem do tempo. Estas probabilidades são representadas por um número real entre 0 e 1, sendo que a soma de todas as probabilidades de transição de um estado para cada um dos demais deve resultar em 1.

As probabilidades de transição das cadeias de *Markov* são representadas através de matrizes de dimensões *n x n*, sendo *n* o número de estados desta cadeia. Estas matrizes são chamadas estocásticas devido ao fato de que cada linha que as compõe é uma distribuição (soma das probabilidades igual a 1). No caso das cadeias de tempo discreto, esta matriz é chamada de matriz de probabilidade de transição, e no caso das cadeias de tempo contínuo ela é chamada de matriz de taxas de transição. No caso das matrizes das cadeias de tempo contínuo, devido aos valores das taxas representarem a freqüência com que as transições ocorrem, a soma dos valores das linhas da matriz não tem sua soma igual a um. Para solucionar esta diferença, a diagonal principal da matriz é composta por valores negativos, fazendo com que a soma das linhas seja zero. Estas matrizes estocásticas apresentam correspondência de uma para um com o diagrama que representa uma mesma cadeia de *Markov* [33].

A análise do processo estocástico é dita estacionária quando analisa as características estatísticas do modelo de maneira independente do tempo *t* em que sua observação é iniciada, ou seja, quando o processo não varia, julgando-se um tempo próximo ao infinito. Já a análise transitória realiza a análise de um estado em função de outro estado prévio, ou seja, a análise probabilística de uma trajetória no espaço de estados.

Através da modelagem do sistema sob forma de uma cadeia de *Markov*, apesar do fato desta não possuir memória com relação a estados anteriores da cadeia, pode-se conhecer as probabilidades de se estar em determinado estado, ou conjunto destes, em determinado momento posterior ao início do processo. É possível também estimar quanto tempo é necessário para atingir-se determinado estado pela primeira vez, e uma série de outras métricas relativas ao sistema e sua evolução.

No exemplo da Figura 11, temos um pequeno modelo de uso utilizando cadeias de *Markov* para a autenticação de usuários numa aplicação. Nesse exemplo o usuário precisa digitar uma senha válida para acessar o menu opções. A cadeia é composta por quatro estados (*Start*, *Passwd*, *PNotOK* e *Menu*) e sete eventos de transição entre esses estados. É possível notar que em cada evento de transição há uma probabilidade associada, a qual determina o possível comportamento do sistema (uso operacional).

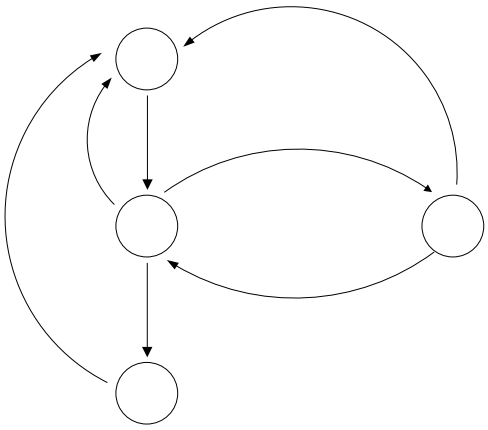


Figura 11: Exemplo de cadeia de *Markov*

3.3.4 REDE DE AUTÔMATOS ESTOCÁSTICOS

Uma rede de autômatos estocásticos (SAN) consiste em um conjunto de autômatos estocásticos individuais que operam de maneira quase independente [35, 36]. Cada autômato é representado por um determinado número de estados, juntamente com regras ou funções de probabilidade que regem os movimentos de um estado para outro do autômato. O estado local de um autômato em determinado tempo *t é* o estado que este autômato ocupa no tempo *t*. Já o estado global da rede de autômatos estocásticos é dado pelo estado local que cada um dos autômatos constituintes ocupa neste mesmo tempo *t* [36]. As taxas de transição de um estado para outro nas redes de autômatos estocásticos são representadas nos próprios arcos do modelo, sendo exponencialmente (no caso de tempo contínuo) ou geometricamente (no caso de tempo discreto) distribuídas.

Uma SAN pode possuir certo grau de interação entre seus autômatos, sendo esta representada pelas taxas funcionais e pelos eventos sincronizantes. No caso de taxas funcionais, que representam uma taxa que varia de acordo com o estado global da rede, estas devem ser avaliadas (calculadas) a cada ocorrência, através da sua fórmula de definição. Os eventos sincronizantes afetam a rede como um todo, fazendo com que uma transição em um autômato dispare simultaneamente uma transição em outros autômatos, sincronizando a interação entre eles.

3.3.4.1 EVENTOS

O estado global de uma SAN pode ser mudado por eventos locais ou eventos sincronizantes. Os eventos locais, como o próprio nome sugere, alteram o estado individual de determinado autômato da rede, alterando o seu estado local. Logo, se tem um novo estado global, cuja

diferença em relação ao anterior se dá em apenas um autômato. Já os eventos sincronizantes podem alterar simultaneamente mais de um estado local, ou seja, promovendo alteração de estado em mais de um autômato ao mesmo tempo.

A Figura 12 ilustra estes dois tipos de eventos. Nela podemos ver os eventos locais *l1, l2 e l3* (autômato A), e o evento local *l4* (autômato B), representando as transições que ocorrem unicamente no seu próprio autômato, representando uma alteração no estado local de cada autômato. Já o evento sincronizante *S,* representa transições disparadas simultaneamente nos autômatos A e B, significando uma alteração no estado global da SAN através da alteração dos estados locais de ambos os autômatos. No autômato A, cabe ressaltar que apenas uma das transições do evento *S* é disparada por vez, conforme a probabilidade de ocorrência assumida.

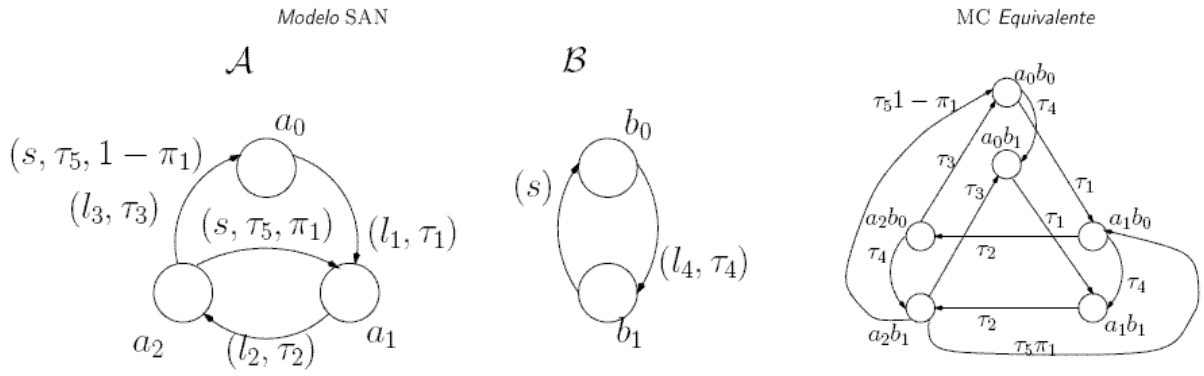


Figura 12: Modelo SAN com eventos sincronizantes

Um evento sincronizante faz com que todos os autômatos afetados por ele disparem uma transição correspondente a este evento. Uma informação importante acerca dos eventos sincronizantes é que, a condição principal para a ocorrência destes eventos é que eles possam ser disparados em todos os autômatos que os contêm. No caso, por exemplo, de um evento estar presente em 3 autômatos, caso em um deles a transição relativa ao evento seja impossível, não é permitido a ocorrência do evento em toda a SAN. Os eventos locais são representados por um identificador e a respectiva taxa do evento. Já as transições representando eventos sincronizantes são identificadas por um nome ou identificador, uma taxa de disparo e a probabilidade de ocorrência do evento. Logo, temos [36]:

- **Nome do evento:** todas as transições relativas a um mesmo evento sincronizante são identificadas pelo mesmo nome.
- **Taxa de disparo:** descreve a taxa em que o evento ocorre, sendo indicada em apenas um autômato. Nos demais autômatos em cujas transições ocorre o evento, a taxa de disparo é omitida. Na Figura 12 a taxa de disparo é mostrada como o segundo item na tripla do evento *S.*

- **Probabilidade de Ocorrência:** caso um evento sincronizante possua mais de uma transição partindo de um mesmo estado local, cada uma dessas transições recebe uma probabilidade de ocorrência, dado que não podem ocorrer simultaneamente. A soma das probabilidades de ocorrência dos arcos de um mesmo evento sincronizante em cada autômato é sempre igual a 1. No caso de um evento possuir apenas uma transição em determinado autômato, a probabilidade de ocorrência é igual a 1, e pode ser emitida. Na Figura 12 a probabilidade de ocorrência é mostrada como terceiro item da tripla do evento *S.*

3.3.4.2 TAXAS

As taxas relativas aos tipos de evento acima citados podem ser de dois tipos: fixa ou funcional. As taxas fixas, como o próprio nome sugere, são representadas por um número real não negativo, não apresentando variação. Já as taxas funcionais representam juntamente com os eventos sincronizantes, os dois modos de interação entre autômatos de uma SAN. Estas taxas não são mais representadas por um número real não-negativo unicamente, mas por uma função discreta dos estados locais de alguns autômatos sobre estes números. Logo, os estados locais da SAN é que indica qual será a taxa utilizada no momento da transição. Na Figura 13, temos, no autômato B, o evento local *l4* apresentando taxa funcional *f*, definida como:

$$f = \begin{cases} \lambda_1 & \text{se o autômato } \mathcal{A} \text{ estiver no estado } a_0; \\ 0 & \text{se o autômato } \mathcal{A} \text{ estiver no estado } a_1; \\ \lambda_2 & \text{se o autômato } \mathcal{A} \text{ estiver no estado } a_2; \end{cases}$$

Assim, o valor da taxa será definido pela avaliação da função *f*, ou seja, em função do estado local atual do autômato A.

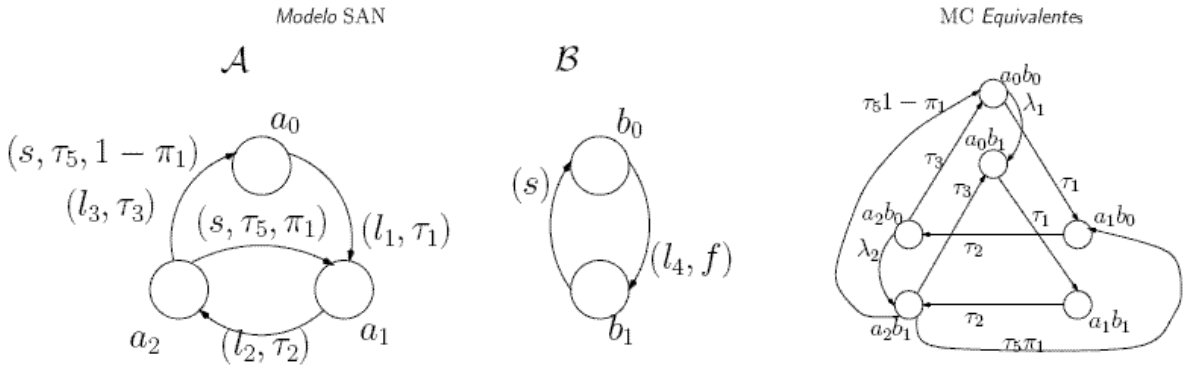


Figura 13: Modelo SAN com taxa funcional

As taxas funcionais não se apresentam apenas em eventos locais, podendo ser utilizadas até mesmo em eventos sincronizantes, inclusive para representar sua probabilidade de ocorrência. A taxa é indicada através da avaliação da função no momento de seu disparo. A vantagem do uso de taxas funcionais frente às taxas constantes é a possibilidade de representação compacta de estruturas complexas.

3.3.5 EXEMPLO DE MODELOS DE USO

O uso de SAN na representação de modelos de uso pode ser exemplificado através das Figuras 14 e 15, onde é apresentado um exemplo utilizado em [1]. A Figura 14 apresenta a interface da aplicação, que é constituída por três janelas de diálogos. A primeira é uma janela de *login* (Figura 14(a)), onde o usuário digita seu *username* e *password*. A segunda é uma janela de mensagem de erro (Figura 14(b)), caso ocorra algum erro na autenticação do usuário. E a terceira é a janela do Menu (Figura 14(c)), onde o usuário pode apenas terminar o aplicativo. O modelo de uso dessa aplicação é descrito em SAN na Figura 15, entretanto não é equivalente ao modelo em cadeia de *Markov* da Figura 11.

Figura 14: Interfaces do sistema de *login*

A estrutura da SAN que descreve o possível comportamento do sistema possui os seguintes componentes:

- Autômatos (2): {*Login Automaton*, *Password Automaton*};
- Estados (6): {*Start*, *Menu*, *Password*} e {*PNotOk*, *Waiting*, *POK*};
- Eventos (5): {*ST*, *QT*, *S*} eventos sincronizantes e {*g*, *f*} eventos locais;

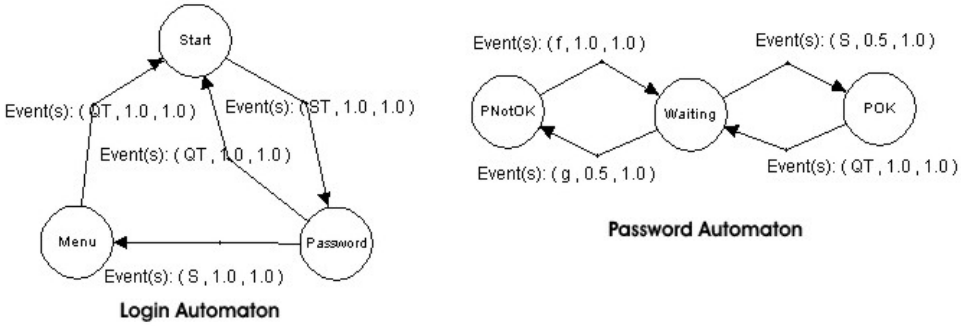


Figura 15: Modelo SAN do sistema de *login*

O comportamento do sistema modelado na Figura 15 pode ser observado através de dois casos de teste para esse modelo, conforme a Figura 16. Nesses casos de teste são apresentados passo a passo, os estados globais da rede e o evento que gerou a transição de estados. É possível notar que sempre que ocorre um evento sincronizante, os estados dos autômatos envolvidos nesse evento são alterados simultaneamente. Já quando ocorrem eventos locais, apenas estados locais ao evento são alterados. Esses casos de teste foram criados em um ambiente integrado para geração de casos de teste e scripts para teste estatístico de software – STAGE [1]. O ambiente STAGE foi desenvolvido no CPTS/PUCRS, em um projeto em colaboração com a HP Brasil. A principal contribuição desse sistema foi o uso desse formalismo para a representação dos modelos de uso e um modelo intermediário (*ISEM* – Modelo do Estado de Interface) para mapear a abstração do modelo de uso com a interface dos componentes do sistema. Dessa forma, a utilização de SAN permitiu uma representação modular de sistemas com complexo comportamento não-deterministico, minimizando a explosão de espaço de estados apresentado em cadeias de *Markov*.

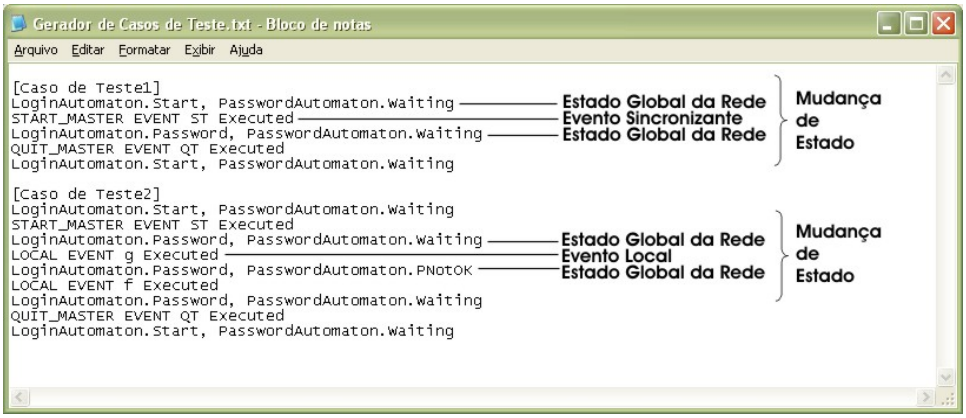


Figura 16: Casos de teste do modelo SAN

Na presente dissertação, busca-se adaptar as funcionalidades do STAGE para comportar a complexidade de programas paralelos, e com isso gerar automaticamente casos de teste e scripts para teste funcional de programas paralelos.

Além do trabalho apresentado em [1], outros trabalhos demonstram a utilização de modelos de uso e métodos estatísticos em etapas de teste de software, seja em programas seqüenciais ou em programas paralelos. Em [37] os autores propõem a utilização das cadeias de *Markov* como um formalismo para a modelagem de uso de aplicações para teste estatístico de software. A utilização do teste estatístico e dos modelos de uso empregando cadeias de *Markov* é embasada na necessidade de métodos estatísticos e modelos de confiabilidade na aplicação do teste funcional. Para tal, é proposto o uso de duas cadeias de *Markov* diferentes nesse processo, uma cadeia de uso e outra de teste. A cadeia de uso consiste no modelo de uso em si, ou seja, a representação dos estados e transições do sistema, juntamente com suas probabilidades de ocorrência. O objetivo desta cadeia é o de apoiar o processo de geração dos casos de teste, possibilitando a aplicação de diversos critérios de cobertura. Já a cadeia de teste é montada a partir dos dados obtidos no processo de teste. A cadeia é alimentada pelos registros de teste, constituindo uma espécie de histórico do processo. Até mesmo as situações de falha passam a fazer parte da cadeia (um diferencial em relação à cadeia de uso), possibilitando inferir a respeito da confiabilidade da aplicação de acordo com a evolução da cadeia, à medida que novas versões da aplicação são testadas, e incorporadas à cadeia.

Em [39], diversos modelos SAN foram criados e comparados com modelos de cadeias de *Markov* em termos de números de estados, escalabilidade e capacidade de leitura desses modelos. Nessa comparação, SAN apresentou-se superior. Em [38], casos de teste foram gerados automaticamente através de modelos SAN. Já em [40], SAN é utilizada na construção de modelos de desempenho aplicados a programas paralelos, representando a comunicação entre processos paralelos com comportamentos síncronos e assíncronos.

Como a proposta dessa dissertação está baseada na utilização de modelos de formalismo discreto para a representação comportamental de aplicações paralelas, fundamenta-se a utilização de SAN para essa abstração, devido a sua capacidade em comportar a complexidade dos programas paralelos [36]. Entretanto, as etapas de teste tornam-se um pouco mais complicadas do que em programas seqüenciais, principalmente devido ao grande número de comunicação e sincronismo necessários. Dessa forma, torna-se indispensável a utilização de técnicas de depuração para a manipulação desses eventos de comunicação [41, 42, 43, 44, 45].

3.4 O PROCESSO DE DEPURAÇÃO

Após as falhas terem sido identificados durante a fase de teste, o propósito do processo de depuração é encontrar e corrigir erros no código do programa que causaram as falhas, sendo considerada uma tarefa típica de programador.

Uma estratégia para isso é a aplicação de abordagens de depuração cíclica. Nessa abordagem, após a detecção da falha, o programador re-executa o programa usando os mesmos dados de entrada, fazendo com que o programa reproduza o mesmo comportamento falho. Esta re-execução acontece sob o controle de uma ferramenta de depuração, o depurador, que permite ao programador executar o programa passo a passo. Após cada passo, a execução do programa pode ser interrompida, e o programador tem a possibilidade de conferir o estado do programa (inspecionando o conteúdo de variáveis, etc), o que permite obter mais informações sobre as razões do comportamento incorreto do programa. Se a informação conseguida durante esta re-execução não for suficiente para encontrar e corrigir os erros, o programador pode executar o programa novamente (com os mesmos dados de entrada), tentando obter mais informações. Este procedimento pode se repetir, até que a informação coletada seja suficiente para localizar e corrigir a falha encontrada.

Para interromper a execução do programa após cada instrução, usa-se *breakpoints*. Estes podem ser inseridos manualmente pelo usuário, e são consideradas funcionalidades importantes, principalmente se o programador já tem idéia de onde possa estar o erro. Dessa forma, o *breakpoint* pode ser inserido a algumas instruções antes da instrução que se suspeita ser responsável pela falha. Assim, quando a execução do programa for interrompida, o usuário pode continuar a execução passo a passo, tentando identificar o erro.

A depuração cíclica é uma abordagem de depuração genérica, podendo ser vista como uma base para o teste e depuração de qualquer programa. No Capítulo 4, mais detalhes sobre depuração serão apresentados, assim como técnicas adicionais necessárias para o teste e depuração de programas paralelos, principalmente programas não-determinísticos, os quais possuem diversas propriedades que dificultam essas etapas.

Capítulo 4

TRABALHOS RELACIONADOS: TESTE E DEPURAÇÃO DE PROGRAMAS PARALELOS

Após ter descrito a área de aplicação, “Computação Paralela” (Capítulo 2), e também os principais conceitos sobre teste e depuração (Capítulo 3), neste capítulo é apresentado em mais detalhes os ciclos de teste e depuração, tanto para programas seqüenciais, quanto para programas paralelos. Iniciando com uma abordagem tradicional sobre teste, no decorrer das Seções serão introduzidos alguns problemas e dificuldades encontradas ao se analisar programas paralelos, o que nos levará a uma revisão sobre os obstáculos da depuração devido aos comportamentos não-determinísticos, assim como possíveis extensões no ciclo tradicional de depuração cíclica. Dessa forma, nesse capítulo serão revisados conceitos importantes que servem como um breve referencial teórico, assim como algumas soluções e suas ferramentas relacionadas.

4.1 CICLO TRADICIONAL DE TESTE E DEPURAÇÃO

As fases de teste e depuração aplicadas no ciclo de vida de um software ocorrem, geralmente, sempre que o final do estágio de implementação é alcançado, onde se obtêm uma versão do software ou componente de software em seu estado executável. Isso significa que todos os erros sintáticos foram removidos, permitindo que o código seja compilado de maneira correta.

O teste e a depuração sempre operam em dois ciclos, onde o programa é executado várias vezes até que um grau suficiente de confiabilidade tenha sido obtido. O ciclo do teste é usado para detectar computações incorretas, enquanto o ciclo da depuração permite obter mais informações sobre os estados e resultados intermediários durante a execução do programa. A idéia principal é identificar, localizar e corrigir os erros existentes no programa, baseado na comparação dos resultados esperados com os resultados observados na execução.

Nesse contexto de teste e depuração, os pesquisadores *Blum* e *Wasserman* [30] apresentaram o termo *simple checking* (Figura 17), como sendo uma abordagem básica para o teste de software, e podendo ser definida como: “seja *f* uma função matemática qualquer, *x*

uma entrada, *y* uma saída e *P* um programa em teste, *simple checking* determina que um programa esteja correto quando *y = f(x)*, ou seja, quando as saídas esperadas (*y*) são iguais às saídas executadas (*f(x)*), identificando uma correta computação da função *f"*. Neste caso, a função *f(x)* descreve o comportamento do programa *P(x)*, que está baseada nas especificações do mesmo.

Para ter certeza de que o programa executa suas funções pretendidas (*correctness*, conforme o Capítulo 3.2.2), é necessário que, durante o ciclo de teste, essa abordagem seja aplicada e verificada para todo o conjunto de entradas X = {*x1,* x*2*, ... }.

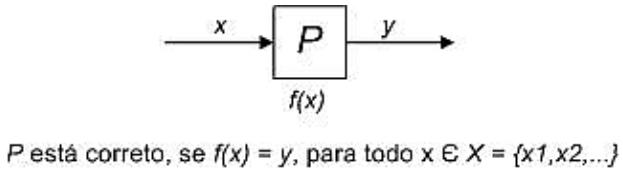


Figura 17: Modelo *simple checking*

Em complemento ao *simple checking*, na Figura 18 é mostrado o ciclo tradicional de teste e depuração. Esse ciclo é representado por um diagrama de fluxo, onde o fluxograma da esquerda representa o ciclo do teste de software, e o da direita o ciclo da depuração. A fase de teste inicia com a seleção dos dados de entrada (casos de teste). Conseqüentemente, os usuários escolhem diversas entradas *(x)* de um conjunto *X* de entradas válidas. Estas entradas são selecionadas para a execução do programa *P*, a qual deve computar um conjunto de saídas *(y)* correspondentes.

Após o término da execução do programa, esta é verificada na busca por falhas que possam ter ocorrido. Essas falhas podem ser: um término inesperado da execução do programa, ou uma computação de um resultado incorreto. Assim, essas duas hipóteses devem ser verificadas, imediatamente após o término da execução do programa. Aqui, situações distintas podem ser observadas, como: a execução do programa pode falhar antes que o programa tenha atingido um estado esperado, ou a execução terminou corretamente. Claro que uma falha é sempre uma indicação de um comportamento incorreto, não importa se gerou um resultado incompleto, ou se não gerou resultado. Conseqüentemente, o ciclo da depuração inicia-se imediatamente logo após essa verificação.

Em situações em que o programa termina de maneira correta, a situação é mais complexa, porque o programa produz certa quantidade de dados de saída, o que parece correto a primeira vista. O problema é que esses resultados devem ser verificados, o que em muitos casos é complicado, dependendo da técnica usada.

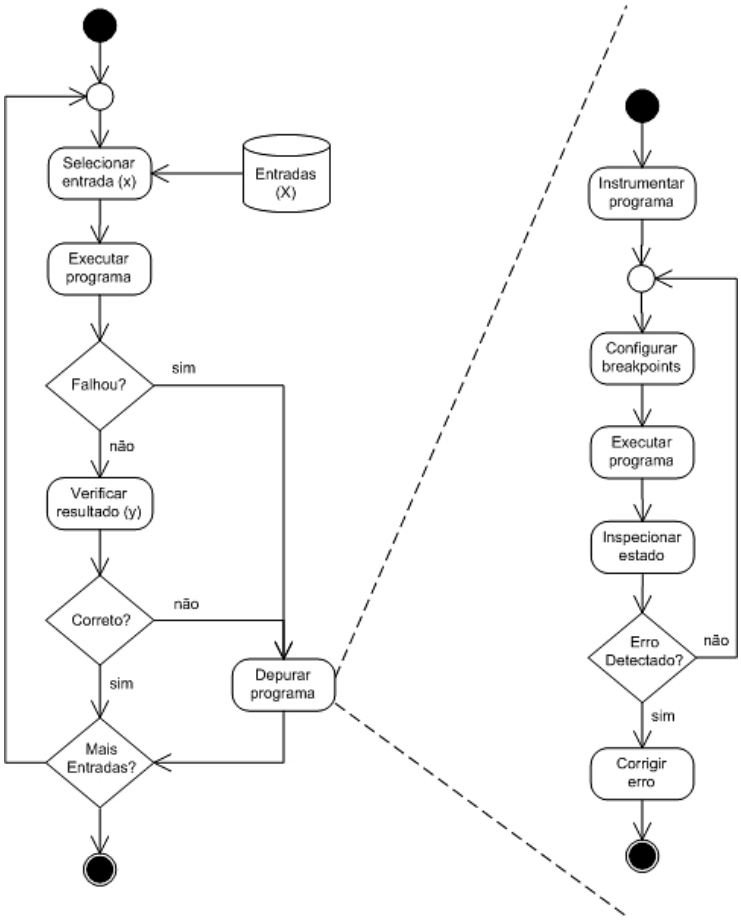


Figura 18: Ciclo de teste e depuração

O uso de técnicas estatísticas foi utilizado em [46], onde o autor utilizou modelos de uso para a verificação dos resultados, através da detecção de erros freqüentes no uso operacional do software. Uma abordagem semelhante foi proposta por [37], onde o autor utilizou métodos estatísticos para a caracterização da execução de programas. Nessa abordagem, as execuções obtidas são sempre amostras do conjunto de possíveis execuções, assim, métodos estatísticos são aplicados para a análise do programa.

Uma diferença importante entre o ciclo do teste e da depuração, é que o teste opera sobre um conjunto de entradas diferentes (*{x1, x2,...}* $\in$ *X*), enquanto a depuração analisa a execução do programa baseado em uma única entrada (*x*) que causou a falha no teste. Dessa forma, o ciclo da depuração termina sempre quando um conjunto particular de dados de entrada, selecionado durante o teste, for avaliado.

Conforme o ciclo da depuração, apresentado no lado direito da Figura 18, a depuração tem inicio com a instrumentação do programa. Isso significa que um código extra é adicionado ao código do programa, o qual serve para observar a execução da aplicação durante a depuração. Com isso, é possível analisar a execução do programa de maneira interativa, visando a identificação da linha do código que contem o erro.

A abordagem tradicional do teste e depuração apresentado nessa seção provê uma breve revisão de suas principais características, e também uma visão geral de sua aplicação em programas seqüenciais (Figura 18). Diversas ferramentas para esse propósito foram implementadas, dando suporte aos usuários nessas etapas [47, 48, 49, 50]. Entretanto, inúmeros pesquisadores aplicam essas técnicas, usadas em programas seqüenciais, para o teste e depuração de programas paralelos, com o objetivo de obter resultados similares. No entanto, existem muitos problemas na depuração de programas paralelos, o que dificulta o uso de abordagens de depuração tradicionais. Esses problemas serão discutidos nas próximas Seções.

4.2 PROBLEMAS DOS DEPURADORES SEQÜENCIAIS

Em relação às questões básicas relacionadas aos depuradores seqüenciais, é possível definir as seguintes restrições:

- O depurador deve afetar o mínimo possível o comportamento da execução de uma aplicação (princípio de *Heisenberg* [51]);
- O depurador deve ser confiável, permitindo que os usuários tenham certeza de suas funções;
- A apresentação das informações do contexto do programa deve ser clara e confiável (variáveis e estados);

As funções do depurador devem comprometer ao mínimo o comportamento da aplicação. Esse princípio de intrusão ou *probe-effect* inserido em sistemas de depuração foi avaliado por alguns pesquisadores, e os erros estabelecidos por esses efeitos foram definidos como *Heisenbugs* [51]. Nesse caso, a intrusão da depuração deve ser mínima, para que se possa garantir a validade dos dados analisados. Em particular às aplicações de depuração, onde o comportamento da execução do programa não deve ser alterado.

Nas etapas de depuração, o depurador viola completamente o princípio de *Heisenberg*. Simplesmente porque o depurador e o programa depurado estão em memória, sendo controlados pelo mesmo sistema operacional, isso pode afetar o comportamento da execução da aplicação de muitas formas, especialmente se o programa for uma aplicação paralela e não-determinística.

Segundo [52], os erros encontrados no teste podem até mesmo ser mascarados, ou não serem percebidos nas etapas de depuração. Isso é possível, visto que a intrusão causada pela inserção de código extra no programa, pode alterar de alguma forma as disposições dos objetos na memória, fazendo com que os *bugs* fiquem mascarados. Isso pode ocorrer até mesmo com um simples *printf* adicionado ao código.

Outro fator importante é que os depuradores devem oferecer informações confiáveis aos usuários, visto que estes utilizam essas ferramentas para observar e tentar inferir como e onde as falhas aconteceram. Qualquer informação incorreta provida pelo depurador guia o usuário em direções erradas, prejudicando, ou até mesmo, impedindo uma correta investigação do erro.

A principal questão relacionada aos usuários dessas ferramentas de depuração é como estas apresentam o contexto das informações, ou os estados intermediários durante as etapas de depuração, como: código fonte, valores de variáveis, informações sobre *threads*, entre outras [51]. Segundo [52], a depuração deve acontecer apenas se puder ser estabelecida uma conexão entre o comportamento observado e o código fonte original, permitindo que este seja monitorado.

4.3 TESTE E DEPURAÇÃO DE PROGRAMAS PARALELOS

Um programa paralelo é considerado basicamente, uma coleção de diversos processos seqüenciais que são executados simultaneamente e se comunicam de alguma forma. Conseqüentemente, as dificuldades básicas encontradas em depuradores seqüenciais, são também evidentes para qualquer depurador paralelo [51]. Entretanto, além dessas dificuldades, os programas paralelos apresentam outros problemas. Em [53], o autor apresenta três justificativas para esses problemas: o aumento da complexidade dos programas paralelos, a quantidade de dados depurados e os efeitos anômalos adicionais.

O aumento da complexidade dos programas paralelos refere-se basicamente ao fato de que em programas paralelos se têm vários processos simultâneos, o que dificulta a estratégia tradicional utilizada em programas seqüenciais [53]. A segunda razão relaciona-se às dificuldades de se observar o que realmente interessa na depuração, evitando o acúmulo desnecessário de informações coletadas. E a última razão diz respeito às dificuldades que ocorrem devido à concorrência e ao sincronismo entre os processos, o que não acontece nos programas seqüenciais [42]. Algumas dessas dificuldades são: *race-conditions* e *probe-effects*. E outras se referem ao não-determinísmo e suas implicações nos programas paralelos como *irreproducibility effect* e *completeness problem*.

Um comportamento não-deterministico é caracterizado por violar o determinismo de execução, ou seja, múltiplas execuções de um mesmo programa, em condições iguais, resultando em diferentes seqüências de eventos sincronizantes e podendo produzir diferentes caminhos de execução [54]. Um exemplo de função de comunicação que introduz o não-determinismo é a função de recebimento de mensagens *MPI_Recv (buffer, any_source)* da biblioteca de funções do *MPI,* onde o parâmetro *any_source* permite o recebimento aleatório de várias mensagens. Dessa forma não é possível determinar qual das mensagens será aceita

primeiro. Isso vai depender da ordem de chegada dessas mensagens, o que caracteriza uma situação não-determinística, e com condição de corrida (*race-condition*).

Existem dois problemas principais causados pelo não-determinismo, o *irreproducibility effect* e o *completeness problem*. O primeiro é caracterizado como sendo a incapacidade de realizar depuração cíclica (*cyclic debugging),* ou seja, a incapacidade de reprodução de um erro encontrado na fase de teste, visto que o programa está sujeito a gerar a cada execução subseqüente, seqüências diferentes de eventos sincronizantes, ou diferentes caminhos de execução. Já o *completeness problem* está diretamente ligado a critérios de cobertura de teste*,* sendo um problema encontrado quando se tenta testar o comportamento de um programa paralelo em todos os caminhos possíveis de execução, o que dependendo do número de caminhos a serem testados pode se tornar impraticável.

Segundo [55], a solução para o *irreproducibility effect,* é a criação de um mecanismo que crie re-execuções equivalentes a uma execução anterior "observada". Esse mecanismo foi chamado de *record&replay,* e é composto por duas etapas. A primeira etapa é a fase de coleta (*record fase*), onde a ordem das mensagens são armazenadas em um histórico de execução (*tracefile*), o qual mantem informações sobre todos os eventos não-determinísticos. E a segunda etapa é a fase de re-execução (*replay*), onde os dados armazenados são utilizados para criar re-execuções equivalentes à execução "observada".

Para o tratamento do *completeness problem,* algumas técnicas também foram criadas como *controlled execution*, *event manipulation* e *artificial replay*. *Controlled execution* é uma abordagem proposta por [56], que oferece um método para a realização de teste automático. Baseada em uma técnica que descreve o comportamento desejado da comunicação entre os processos, o programa paralelo é executado em um comportamento forçado, evitando assim situações de condição de corrida. Uma outra abordagem dessa técnica foi proposta por [57], onde padrões de controle são aplicados para estabelecer as ordens entre os eventos de comunicação. Tais padrões são regras dinâmicas que definem a ordem de interação entre os processos.

Outros trabalhos tratam o *completeness problem* mais diretamente, conforme pode ser visto em [42, 53]. Aqui os autores apresentam uma técnica chamada *event manipulation* e *artificial replay*. A idéia básica consiste em coletar informações referentes aos eventos de comunicação sincronizantes em uma execução inicial da aplicação, e assim identificar todas as condições de corrida existentes. Com isso, em uma segunda etapa, re-execuções artificiais são criadas através da manipulação das ordens dos eventos, permitindo que a aplicação tenha um outro comportamento. Os autores afirmam que conseguem investigar facilmente diferentes execuções para um mesmo conjunto de entradas. Além disso, todas as combinações possíveis de execução do programa podem ser criadas através da manipulação dos eventos, caso todas as combinações das mensagens de recebimento forem testadas. Inicialmente esse

processo era manual [53], mas em [42] ele foi automatizado sem a necessidade de interação de usuários.

Algumas outras técnicas foram propostas para solucionar os problemas do não-determinismo, mas a maioria está baseada em *record&replay.* Como o processo de coleta e armazenamento das informações do programa paralelo é feito com a utilização de monitores, um monitor é basicamente um código extra (sonda) adicionado ao código fonte do programa, visando coletar informações. Essa estratégia também é conhecida como "observar um programa". O uso de monitores introduz o *probe-effect,* ou *overhead* causado pela intrusão do processo de monitoração. Isso significa que a coleta de informações em tempo de execução pode influenciar o resultado do programa, seja na alteração dos tempos ou na ordem dos eventos.

Na próxima seção um resumo das principais ferramentas existentes para a construção e análise de programas paralelos será apresentado. Inicialmente um modelo genérico para a classificação dessas ferramentas será mostrado. Logo após, estas ferramentas serão classificadas de acordo com suas principais características.

4.3.1 MODELO E ARQUITETURA DE FERRAMENTAS PARALELAS

Segundo [58], qualquer ferramenta de análise de programas paralelos consiste em dois itens principais, um componente de observação (monitoração) e um componente de análise. Outra característica importante é como e quando esses componentes interagem com a aplicação, sendo classificados em dois grupos: *on-line* e *off-line*.

Quando a interação entre os componentes ocorre de maneira *on-line,* os dados do monitor são transportados durante a execução do programa para o componente de análise. Já no caso *off-line*, os dados necessários para análise são coletados pelo monitor enquanto o programa é executado e são armazenados em um histórico de execução (*tracefile*) para serem analisados somente após o término do programa.

Ambas as abordagens tem características diferentes. Inicialmente métodos *on-line* são mais flexíveis devido à capacidade de poder determinar e inspecionar cada estado durante a execução do programa. Por outro lado, os métodos *off-line* permitem algumas facilidades, principalmente porque ferramentas *on-line* apresentam apenas estados referentes ao passado e ao presente, enquanto ferramentas *off-line* permitem uma análise completa dos estados da aplicação, feita do início até o fim da execução do programa. Um exemplo dessa diferença pode ser percebido em técnicas para detecção de condições de corrida, onde métodos *on-line* são incapazes de identificar um conjunto completo de condições de corrida durante a execução do programa.

A maioria das ferramentas de monitoração são partes integrantes de ferramentas de análise, especialmente as ferramentas de análise *on-line*. Como exemplo é possível citar algumas ferramentas de depuração como *P2D2* [59], *PDBG* [60] e *PDT* [61] que utilizam o *GNU debugger – gdb* como depurador de baixo nível. Nesses casos, o monitor é completamente integrado no ambiente de depuração.

Já quanto à capacidade de leitura dos históricos de execução (*logs* ou *tracefiles*) gerados pelas ferramentas de monitoração, a maioria dos formatos é em *ASCII*, o que oferece um alto grau de flexibilidade e leitura. Mas essa facilidade nem sempre é favorável, principalmente em casos em que a quantidade de dados monitorados é muito grande. Para esses casos, a solução encontrada foi criar *logs* em formatos binários, e muitas vezes de maneira compactada, diminuindo dessa forma o seu tamanho e permitindo o armazenamento do comportamento dos programas de forma mais eficiente.

4.3.2 RESUMO DAS PRINCIPAIS FERRAMENTAS EXISTENTES

Baseado no modelo, nas características e nos problemas apresentados nas Seções anteriores, diversos trabalhos introduzem soluções inovadoras para tais problemas. Essas soluções correspondem ao estado da arte no contexto de ferramentas para criação, manipulação e análise de programas paralelos.

Em ambientes de teste e depuração existem trabalhos relacionados à descrição de ambientes completos que incluem a depuração como parte integral de suas estratégias de desenvolvimento. Alguns trabalhos discutem as vantagens dessa estratégia [62]. Em [45] é descrito uma combinação entre depuradores distribuídos – *DDBG* [60] com ferramentas de teste estrutural – *STEPS* [63]. Essa estratégia permite localizar erros através de análises simbólicas do código realizado com inspeções do código em execuções controladas. Uma estratégia parecida é oferecida por [56] em um ambiente integrado de ferramentas automatizadas para análise de programas que utilizam padrões de comunicação baseados em *PVM*.

Outros exemplos de ambientes integrados podem ser vistos em *GRADE* (*Graphical Application Development Environment*) [64], e *MAD* (*Monitoring and Debugging Environment*) [65]. A idéia básica em *GRADE* é prover um suporte gráfico de alto nível para programação baseada em uma linguagem gráfica – *GRAPNEL* para ambientes *PVM*. Esse ambiente oferece módulos para a construção, execução, depuração, monitoração e visualização de programas paralelos baseados em troca de mensagens, podendo ser acessado pelos usuários através de uma interface gráfica. Como ferramenta de depuração, *GRADE* integra o *DDBG* como depurador distribuído.

Outro forte ambiente integrado que combina um conjunto de ferramentas para monitoração e análise de programas paralelos é o *MAD* [65]. De acordo com um modelo genérico para ferramentas de análise, os autores classificam os componentes de *MAD* em módulos de monitoração e análise. Diversas possibilidades são oferecidas em termos de monitoração. Uma implementação inicial chamada *EMU* (*Event Monitoring Utility*) gera históricos de execução para análise *off-line* em um formato de dados proprietário [42]. Uma das idéias principais é usar essa ferramenta para medir os efeitos causados na execução da aplicação com as técnicas de monitoração empregadas. Com isso, diversas estratégias foram implementadas para correção do *overhead* (intrusão) causado pelos monitores, objetivando remover a perturbação tanto no tempo de ocorrência dos eventos quanto em suas ordens.

Outra ferramenta pertencente ao *MAD*, e que se integra ao *EMU*, chama-se *PARASIT* (*Parallel Simulation Tool*), responsável por gerar execuções equivalentes em programas paralelos não-determinísticos. *NOPE* (*Nondeterministic Program Evaluator*) que representa um mecanismo completo para a técnica de *record&replay* é outra ferramenta integrada ao ambiente [42]. Ela apresenta uma característica adicional, a possibilidade de gerar manipulações automáticas de eventos com o objetivo de solucionar o *completeness problem,* conforme apresentado nas Seções anteriores. Além dessas ferramentas, *MAD* apresenta ainda um módulo central – *ATEMPT* (*A Tool for Event Manipulation*), capaz de visualizar execuções paralelas utilizando diagramas baseados em técnicas de tempo e espaço (*space-time diagrams)* [43].

Para a representação gráfica do comportamento de programas paralelos, existem diversos trabalhos que descrevem abstrações de modelos comportamentais em diversos tipos de diagramas. Segundo [66], uma das melhores formas é através de Redes de *Petri* , o que pode ser aplicado em diferentes níveis de abstração de programas com características diferentes. Outro tipo de representação gráfica é o grafo de dependência (*dependency graph).* Ele é capaz de caracterizar visualmente dependências de dados e controle de operações executadas por um programa [67].

Existem outros tipos de representações gráficas, mas um dos mais usados é o diagrama de tempo e espaço, que foi introduzido por [68] para expressar ordem entre os eventos distribuídos. Esse diagrama é usado em muitas ferramentas para depuração paralela e ajuste de desempenho [69]. Um bom exemplo do uso desse diagrama é *Paragraph*, que possui além de diagramas de tempo e espaço, muitas outras formas de visualização [70]. Outra ferramenta que incorporou o diagrama de tempo e espaço foi *AIMS* [71]. *AIMS* é baseado em instrumentação em nível de código e provê portabilidade entre diferentes tipos de hardware. Uma ferramenta similar a *AIMS* é *XPVM* [72], e possui uma interface gráfica para análise de programas paralelos baseados em *PVM,* abstraindo graficamente tempos de computação, *overheads* de comunicação e tempos de espera, podendo ser visualizados tanto de maneira *on-line* quanto *off-line.* Uma comparação similar para programas *MPI* pode ser vista em *Upshot*

[73] e *Vampir* [74], e mais recentemente em *Jumpshot4* [75]. Existem outros trabalhos com propostas semelhantes às citadas acima, entretanto uma revisão exaustiva foge ao objetivo do presente trabalho.

Essa seção apresentou os trabalhos mais importantes e que servem de base para a definição da estratégia que está sendo proposta. Na próxima seção, o ambiente de teste implementado, será apresentado. Suas funcionalidades serão exemplificadas, assim como o levantamento dos problemas encontrados.

Capítulo 5

AMBIENTE IMPLEMENTADO: ENGINE DE TESTE

Após ter descrito a área de aplicação desse trabalho, “Computação Paralela” (Capítulo 2), e também os principais conceitos sobre teste e depuração (Capítulos 3 e 4), aqui será apresentado o ambiente de teste que foi implementado para dar suporte à estratégia proposta. Esse ambiente será detalhado e suas funcionalidades serão avaliadas assim como as dificuldades encontradas na avaliação dos resultados.

5.1 VISÃO GERAL DO AMBIENTE

Assim como em outros processos de teste de software, o processo utilizado nesse trabalho consiste basicamente em aplicar um conjunto de casos de teste na execução de uma aplicação, e verificar se há inconsistência entre o resultado obtido e o resultado esperado. A diferença é que, nesse trabalho, as aplicações a serem testadas são aplicações paralelas, e conforme apresentado no Capítulo 4, exigem muito mais cuidados nas etapas de teste do que os programas seqüenciais.

Além do não-determinismo, o qual possibilita que uma aplicação paralela apresente caminhos de execução diferentes, existe também o fato de estarmos tratando de programas com diversos processos simultâneos e com a ausência de um relógio global, o que torna indispensável a utilização de técnicas de depuração e monitoração, principalmente para a identificação dos estados atingidos em cada processo envolvido durante a execução do programa a ser testado.

Uma vez que a proposta deste trabalho é uma estratégia para diminuir a intrusão do teste de software em programas paralelos, um ambiente de teste foi desenvolvido, e teve sua proposta formalizada e apresentada na Conferência Latino Americana de Informática (CLEI 2005), conforme Anexo A.

Para a implementação desse ambiente de teste, foi definido um processo o qual define as etapas e técnicas utilizadas, além de um ciclo de teste, que apresenta mais detalhes às

abordagens de cada etapa envolvida. A técnica de teste utilizada é a técnica de teste de software funcional baseada em modelos de uso. Ela, juntamente com um modelo de falhas (que será apresentado na Seção 5.3.4), visa testar o comportamento do programa, buscando falhas na comunicação entre os processos. Para a representação dos modelos de uso, adotou-se SAN como o formalismo capaz de suportar a complexidade dos programas paralelos.

Nas próximas seções, o processo de teste será apresentado, assim como a metodologia utilizada, as principais funcionalidades, além das dificuldades encontradas na avaliação dos resultados.

5.2 METODOLOGIA DE TESTE

O processo de teste é mostrado na Figura 20. As técnicas utilizadas nesse processo norteiam as contribuições mais importantes do ambiente de teste, e estão baseadas na geração automática de casos de teste e scripts para teste de software funcional, bem como, na criação de um módulo de análise *on-line* que implemente o teste funcional baseado nos estados do modelo (casos de teste).

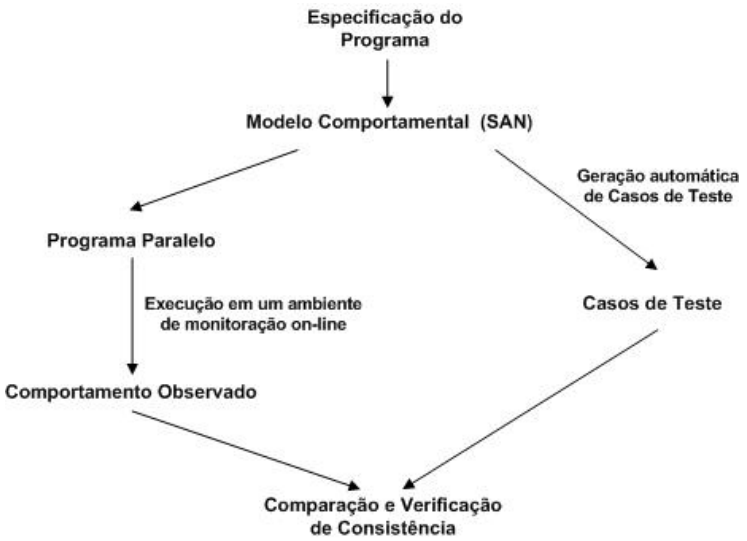


Figura 19: Processo de teste funcional baseado em modelos SAN

Conforme a Figura 19, a geração dos casos de teste é feita automaticamente através do modelo SAN, como sendo um possível ou esperado comportamento operacional da aplicação em teste. Com essa abordagem, busca-se criar através das estimativas probabilísticas do modelo, casos de teste não-determinísticos.

Para a implementação do processo de teste definiu-se um ambiente integrado conforme a Figura 20. O ambiente é composto por três módulos principais: módulo gerador

de casos de teste e scripts, módulo de análise e módulo de monitoração e execução, todos desenvolvidos em JAVA. O módulo gerador de casos de teste e scripts estende os conceitos utilizados em [1].

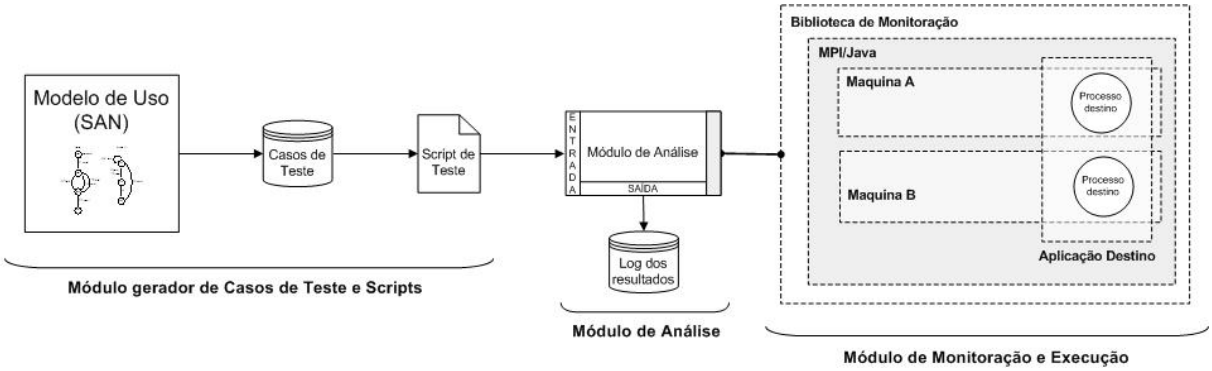


Figura 20: Ambiente integrado de teste

O módulo de análise juntamente com o módulo de monitoração e execução, compõem o *engine* de teste, e são eles os responsáveis pelo teste funcional da aplicação. Conforme a Figura 20, o módulo de análise recebe como estímulo (entrada), um script de teste contendo casos de teste e parâmetros de execução. A estrutura do script será apresentada na próxima seção.

A interação entre os módulos do *engine* e a aplicação em teste é feita de maneira *on-line*, onde os dados monitorados são transferidos para o módulo de análise em tempo de execução. Esses dados são coletados através de uma biblioteca de funções de monitoração, que é associada à execução do programa no instante da instrumentação do código da aplicação.

Com a integração desses módulos, é possível monitorar e analisar a execução da aplicação através dos estímulos oferecidos pelo script de teste, testando o comportamento do programa na busca por situações de inconsistência entre os estados atingidos na execução e os estados do modelo. Assim, sempre que for identificada uma inconsistência (falha) entre os estados e eventos testados, se fixa essa falha, a qual é reportada em um histórico da execução (*tracefile*), até o estado de inconsistência atingido, tendo-se assim, um conjunto de estados intermediários que ajudem na identificação do erro que causou a falha identificada.

5.3 PRINCIPAIS FUNCIONALIDADES

As funcionalidades do ambiente de teste ficam mais explícitas através do ciclo descrito na Figura 21. Esse ciclo apresenta o fluxo completo do processo de teste, que tem inicio na especificação do programa, e na modelagem deste usando SAN. A partir dessa etapa, o ciclo

se divide em dois fluxogramas. O fluxograma da esquerda representa a etapa de geração de casos de teste, e termina com a criação dos scripts. Já o fluxograma da direita, representa a etapa de teste, a qual inicia com a codificação e instrumentação do programa, ambos baseados no modelo SAN, seguidos pela leitura do script de teste, o teste propriamente dito, e terminando com o log do resultado do teste.

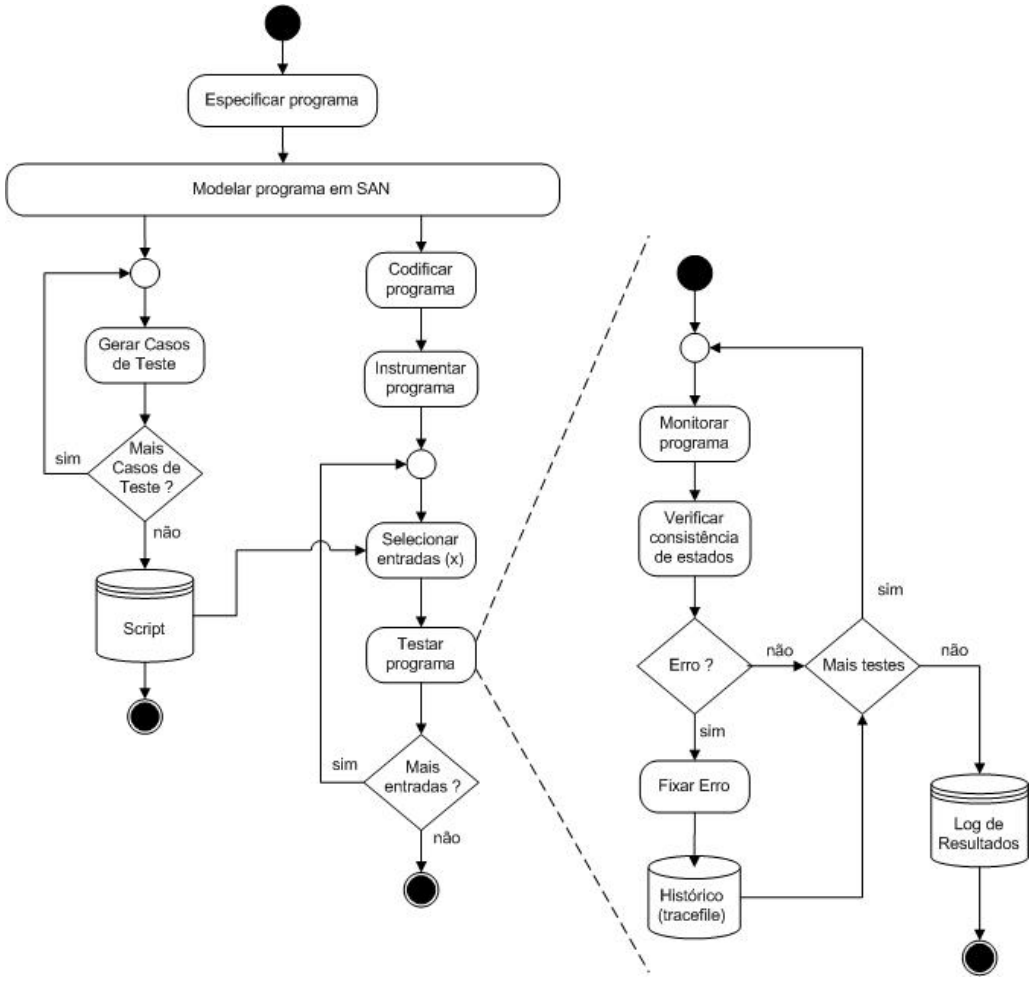


Figura 21: Ciclo de teste

Nas próximas seções, as principais funcionalidades serão explicadas, objetivando apresentar mais detalhes as técnicas utilizadas.

5.3.1 MODELAGEM DA APLICAÇÃO

Nessa etapa, a especificação do programa é usada para a modelagem da aplicação em SAN. A estrutura de uma SAN foi apresentada no Capítulo 3, e é composta por diversos autômatos, onde cada autômato corresponde a um processo da aplicação modelada.

Como exemplo, utilizaremos uma aplicação paralela não determinista para ordenação de vetores. Como o próprio nome diz, esse programa consiste em diversos processos, onde um destes (o mestre) fica responsável por controlar a distribuição e o recebimento dos vetores a serem ordenados pelos outros processos (escravos), os quais ordenam os vetores recebidos e os devolvem para o mestre.

O modelo SAN dessa aplicação é apresentado na Figura 22. Esse modelo é composto apenas por três autômatos, um mestre e dois escravos. Conseqüentemente, o código desse modelo deve ser implementado para executar em três processos.

Após a criação do modelo, as probabilidades de ocorrência devem ser inseridas, as quais determinam a ordem de acontecimento dos eventos, e conseqüentemente, a ordem dos estados globais atingidos pelo modelo. Na Figura 22, esses valores não estão explicitados.

Para auxiliar na geração dos casos de teste e scripts, algumas convenções foram estabelecidas, e podem ser verificadas na própria Figura 22:

- Cada autômato terá seu estado inicial aleatório, a menos que contenha um estado do tipo s*tart*, sendo identificado visualmente através de uma seta ao lado esquerda do estado (MESTRE.not_set, ESCRAVO1.not_set, ESCRAVO2.not_set).
- É necessário que apenas um autômato tenha um estado do tipo *start_master*, o qual iniciará primeiro que os demais (MESTRE.not_set).
- É necessário também que algum autômato tenha um estado final (*quit_master*) (estado *QUIT* de todos os autômatos).
- Apenas os eventos de início, fim e sincronizantes devem ser do tipo *master* ou *slave*. O restante dos eventos deve ser local.
- Eventos do tipo *master* chamam eventos do tipo *slave*, alterando os estados de ambos os autômatos de maneira sincronizante. Eventos locais alteram apenas estados locais de autômatos.
- Cada autômato possui um conjunto de estados locais, e o estado global da rede é identificado através do conjunto de estados locais de cada autômato.

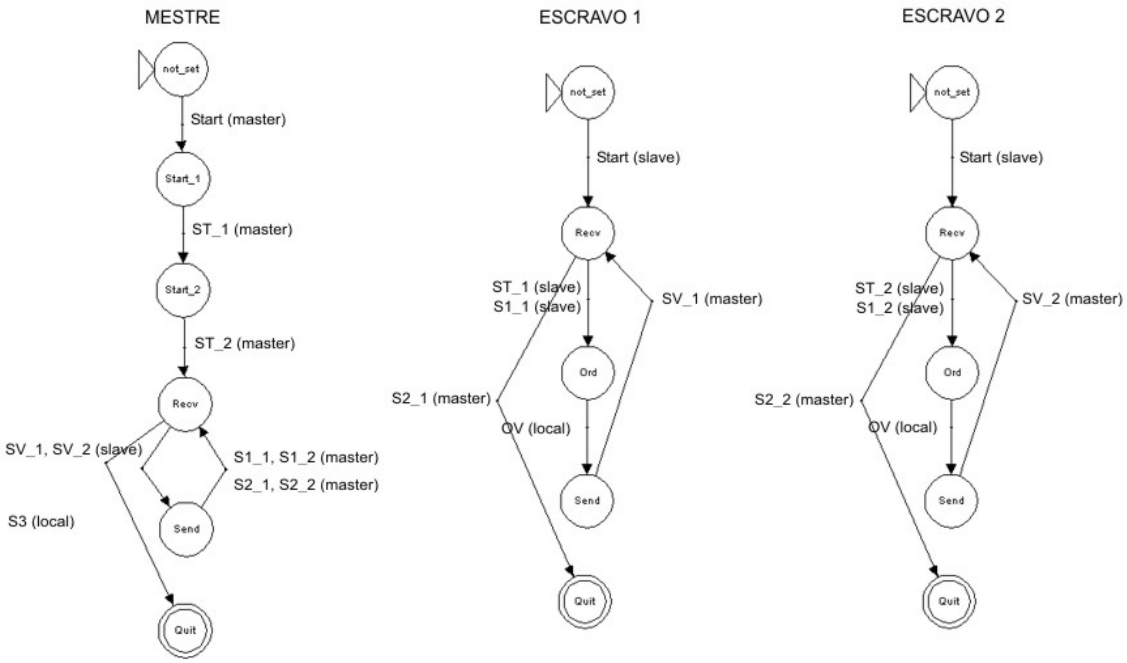


Figura 22: Modelo SAN para ordenação de vetores

A descrição dos eventos de transição de estados do modelo é a seguinte:

- **MESTRE**
  - **Start (master)**: evento para inicialização do processo.
  - **ST_1, ST_2 (master)**: evento sincronizante para envio de um vetor desordenado para um escravo (MPI_Ssend).
  - **SV_1, SV_2 (slave)**: evento sincronizante para recebimento não-determinístico de um vetor já ordenado por um escravo (MPI_Recv).
  - **S1_1, S1_2 (master)**: evento sincronizante para envio de um novo vetor desordenado para um escravo (MPI_Ssend).
  - **S2_1, S2_2 (master)**: evento sincronizante para envio de mensagem terminado comunicação com um escravo (MPI_Ssend).
  - **S3 (local):** evento local indicando que aplicação terminou.

- **ESCRAVOS (1 e 2)**
  - **Start (slave)**: evento para inicialização do processo.
  - **ST_1, ST_2 (slave)**: evento sincronizante para recebimento de um vetor desordenado do mestre (MPI_Recv).
  - **OV (local):** evento local (função para ordenação de vetores).
  - **SV_1, SV_2 (master)**: Evento sincronizante para envio de um vetor já ordenado para o mestre (MPI_Ssend).

- **S1_1, S1_2 (slave)**: evento sincronizante para recebimento de um novo vetor desordenado do mestre (MPI_Recv).
- **S2_1, S2_2 (master)**: evento sincronizante para recebimento de mensagem terminado comunicação com o mestre (MPI_Ssend).

5.3.2 GERAÇÃO DE CASOS DE TESTE E SCRIPTS

Logo após o modelo SAN ter sido criado, e as probabilidades terem sido atribuídas, gera-se então os casos de teste. Estes são gerados aleatoriamente cobrindo todos os arcos (eventos) do modelo, seguindo a ordem das probabilidades de ocorrência atribuídas na etapa anterior. Com os casos de teste criados, gera-se o script de teste, onde são incluídos apenas os casos de teste não repetidos.

Como a técnica de teste utilizada é a técnica de teste funcional baseada em estados do modelo, cada caso de teste contem uma seqüência de estados globais e eventos, os quais têm sua atingibilidade testada na execução da aplicação. Para isso, o script de teste foi definido num formato de arquivo XML, conforme a Figura 23.

```
1  <?xml version='1.0' encoding='us-ascii'?>
2  <!DOCTYPE script_engine SYSTEM "script.dtd">
3
4  <script_engine naut="3">
5
6      <processors>
7          <procalias alias="MESTRE" id="1"/>
8          <procalias alias="ESCRAV01" id="2"/>
9          <procalias alias="ESCRAV02" id="3"/>
10     </processors>
11
12     <parameter name="TIME_OUT" value="2"/>
13
14         <test_case id="1">
15                 <global_state>
16                         <local_state procid="1" stateid="not_set"/>
17                         <local_state procid="2" stateid="not_set"/>
18                         <local_state procid="3" stateid="not_set"/>
19                 </global_state>
20                 <event name="start" procid="1" type="SYN"/>
21                 <global_state>
22                         <local_state procid="1" stateid="Start_1"/>
23                         <local_state procid="2" stateid="not_set"/>
24                         <local_state procid="3" stateid="not_set"/>
25                 </global_state>
26                 <event name="S2_1" procid="2" type="SYN"/>
27                 <final_state>
28                         <local_state procid="1" stateid="Quit"/>
29                         <local_state procid="2" stateid="Quit"/>
30                         <local_state procid="3" stateid="Quit"/>
31                 </final_state>
32         </test_case>
33
34 </script_engine>
35
36
```

Figura 23: Script XML

No script são definidos o número de autômatos (linha 4), um nome para o mapeamento entre os autômatos do modelo e os processos da aplicação (linhas 7-9), um tempo máximo de espera do teste (*time-out*, linha 12), e um conjunto de casos de teste gerados. Em cada caso de teste (linhas 14 – 32) devem estar definidos os dados a serem testados (conjunto de estados globais e eventos), os quais devem obedecer a seguinte ordem: ***global_state*** + ***event*** + ***global_state,*** devendo o último estado do caso de teste ser um ***final_state*.**

É importante lembrar que, como na geração dos casos de teste os autômatos são percorridos de um estado inicial até um estado final, cada caso de teste contem uma seqüência de dados equivalente a um caminho de execução. Conseqüentemente, um script formado por diversos casos de teste contem, na verdade, um conjunto de caminhos esperados no uso operacional da aplicação em teste. E quando se testa o programa, verifica-se se o caminho percorrido na execução é igual ao caminho contido no caso de teste.

5.3.3 CODIFICAÇÃO E INSTRUMENTAÇÃO

Como todo o ambiente de teste foi implementado em JAVA, optou-se por usar JAVA e MPI/JAVA, para a codificação da aplicação paralela, e também para a biblioteca de troca de mensagens, respectivamente. Tanto a codificação quanto a instrumentação da aplicação devem basear-se no modelo SAN. A Figura 24 apresenta um trecho do código de uma aplicação, já instrumentado. Nele, é possível notar que são inseridas algumas funções de monitoração (linhas 4, 12 e 14), através da função "*diretivas*.*reportState*()". Essa função é um método do *engine* de teste, capaz de identificar a ocorrência de eventos em tempo de execução, e logar os estados atingidos pelos processos envolvidos. Tal função deve ser inserida na etapa de instrumentação do código, e possui os seguintes parâmetros:

- ***diretivas.reportState*(*"State", "Event", "Type"*)**

Esses parâmetros indicam, através do evento ocorrido (*Event*), qual o estado que está sendo atingido (*State*), o tipo de evento (*Type* - sincronizante (*SYN*) ou local (*LOC*)), e em qual processo esse evento ocorreu. Dessa forma, essa é a abordagem utilizada para monitorar o comportamento da aplicação em teste, o qual é identificado automaticamente em tempo de execução.

```
1  Mensagem aux[] = new Mensagem[1];
2  int i = 0;
3
4  diretivas.reportState("Start_1", "Start", "SYN");
5
6  for(i=2; i<num_procs; i++){
7      if(i<=vetor.length+1){
8          aux[0] = vetor[i-2];
9          try{
10             MPI.COMM_WORLD.Ssend(aux, 0, 1, MPI.OBJECT, i, 0);
11             if(i != (num_procs-1)){
12                 diretivas.reportState("Start_"+i, "ST_"+(i-1), "SYN");
13             }else{
14                 diretivas.reportState("Recv", "ST_"+(i-1), "SYN");
15             }
16             enviados++;
17         }catch(MPIException e){
18             e.printStackTrace();
19         }
20     }
21 }
```

Figura 24: Exemplo de monitores

Em termos de arquitetura e implementação, o *engine* de teste é instanciado em um processo que roda em paralelo aos processos da aplicação. Assim, sempre que algum processo chama o método “diretivas.reportState()”, este na verdade está enviando uma mensagem para o processo do *engine*, a qual contem os dados referentes ao evento e estado atingido. Dessa forma, o *engine* fica constantemente recebendo essas mensagens e salvando os estados globais atingidos no histórico de execução.

5.3.4 TESTE (ANÁLISE)

Para a etapa de teste, o script é lido pelo *engine*, e os casos de teste são armazenados em memória. Com isso, a aplicação é executada e monitorada, e os estados de execução atingidos são identificados e comparados automaticamente com os estados e eventos lidos do caso de teste. O teste é considerado bem sucedido, quando o caso de teste aplicado identifica alguma falha, ou seja, quando ocorre alguma inconsistência entre os estados testados.

Um modelo de falhas foi definido baseado nas falhas tradicionais de sistemas paralelos e distribuídos. Como exemplo dessas falhas, temos: *crash faults*, *fail-stop*, *omission faults*, *delay faults*, e *bizantine faults*. Entretanto, em abordagens de teste funcional (*black-box*), que utilizam modelos de uso para a geração automática de casos de teste, os modelos de falhas geralmente baseiam-se na cobertura dos possíveis estados atingidos a partir do modelo.

Como o foco desse trabalho é aplicar o teste funcional para testar o comportamento de aplicações paralelas baseadas em troca de mensagens, e também pelo fato do ambiente de

execução (*Cluster*/*MPI*) ser considerado um ambiente "comportado" ou facilmente controlado, e assim menos propenso a falhas, o modelo assumido busca as seguintes falhas:

- Incapacidade de atingir estados definidos no modelo.
- Atingibilidade de estados não permitidos pelo modelo.
- Possíveis *deadlocks* e *livelocks* (*time-out*).
- Baixo desempenho (gargalos de comunicação).
- Falhas de omissão.

Inicialmente, falhas vinculadas a atingibilidade de estados, são automaticamente identificadas. Demais tipos de falhas, devem ser verificadas manualmente através do log de resultados do teste, que contêm o conjunto de estados e eventos atingidos na execução até o instante da falha.

5.4 RESULTADOS E EXPERIMENTOS

5.4.1 DEFINIÇÃO DO EXPERIMENTO E AVALIAÇÃO DOS RESUTADOS

Nessa seção, um experimento será feito visando validar as funcionalidades do ambiente de teste e identificar problemas relacionados. A aplicação utilizada para isso será a ordenação de vetores, já mencionada nas Seções anteriores. E o experimento consiste basicamente em aplicar o processo de teste nessa aplicação paralela.

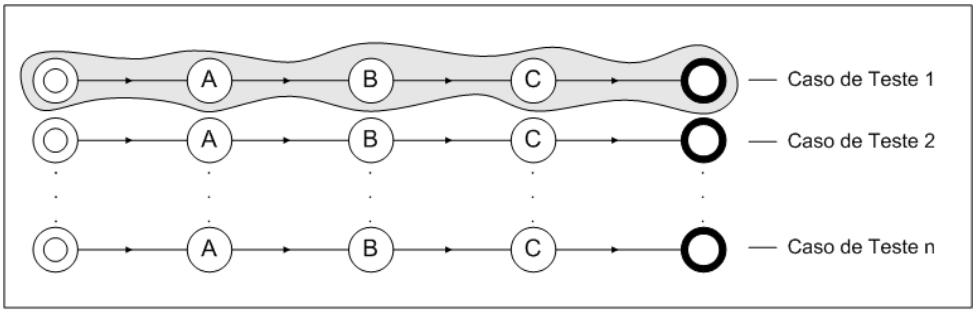
Para validar as **funcionalidades de monitoração e análise**, foram criados manualmente (para evitar problemas relacionados ao não-determinísmo), casos de teste contendo seqüências de dados de teste, os quais se tinham certeza que seriam atingidos na execução da aplicação. Buscou-se com isso, evitar problemas com o não-determinismo, e garantir dessa forma, que os estados e eventos contidos nos casos de teste fossem atingidos na execução da aplicação, ou seja, que o comportamento esperado nos casos de teste fosse igual ao comportamento monitorado.

Para se ter essa certeza, diversos scripts foram criados contendo inúmeros casos de teste, e todos tiveram 100% de atingibilidade. Com isso, é possível validar as abordagens de monitoração e instrumentação utilizadas no *engine*. Entretanto, esse experimento inicial serviu para validar as funções de monitoração, buscando provas de consistência na atingibilidade de estados.

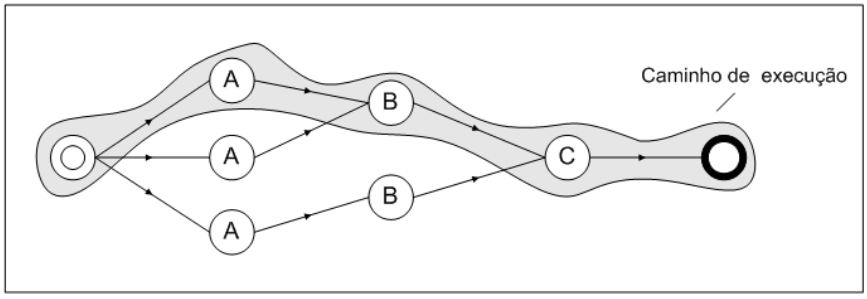
5.4.2 PROBLEMAS ENCONTRADOS

Os problemas surgiram quando se iniciaram os testes para verificar a capacidade do *engine* em identificar falhas, ou seja, durante a etapa de validação do processo de teste, onde casos de teste foram gerados automaticamente a partir do modelo SAN. Essas dificuldades foram sentidas quando os resultados dos testes indicavam uma presença muito grande de falhas, mesmo em casos de teste que não deveriam encontrar falhas, caracterizando uma situação de "falsos *bugs*".

A análise desses "falsos *bugs*", nos levou a descobrir que o problema que estávamos tendo, era um problema já enfrentado por muitos pesquisadores da área na tentativa de solucionar dificuldades relacionadas ao não-determinísmo. Algo que muitas vezes pode ser solucionado com técnicas baseadas em *record*/*replay*, conforme já foi mencionado no Capítulo 4. Entretanto, o que nos impediu de utilizar essas abordagens, é que o foco do nosso problema está na geração dos casos de teste, pois geramos casos de teste determinísticos para testar aplicações não-determinísticas.



A – Diversos casos de teste (cada caso de teste é um conjunto de estados e eventos de transição, representando um modelo de uso, ou um possível uso operacional do software modelado)



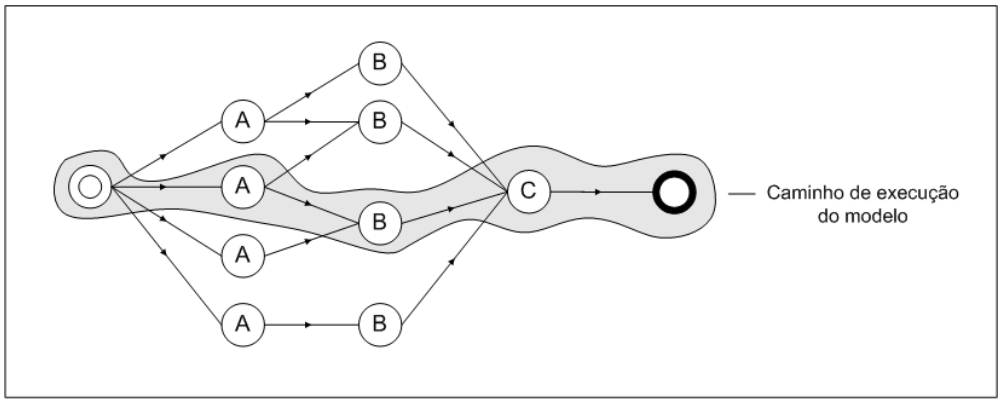
B – Caminho de execução da aplicação paralela (diversos caminhos possíveis de execução devido ao não-determinísmo)

Figura 25: Casos de teste determinísticos

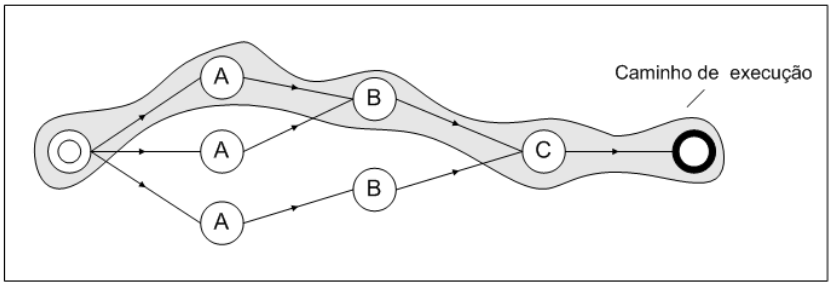
A ocorrência desses "falsos *bugs*" dá-se, pois para cada caso de teste a ser testado (Figura 25 - A), executa-se a aplicação, e testa-se a consistência entre os estados atingidos na execução (Figura 25 - B), e os estados contidos no caso de teste. Algo que seria correto em

aplicações com comportamento determinístico, visto que é possível testar a aplicação várias vezes com um mesmo caso de teste, que o resultado provavelmente será sempre o mesmo. Entretanto, como em aplicações paralelas a cada execução podemos ter comportamentos diferentes (caminhos de execução diferentes), conclui-se que essa abordagem para geração de casos de teste não suporta programas paralelos não-determinísticos, o que dificulta a validação do processo de teste, ou seja, a interpretação dos resultados do teste.

Como solução para esse problema, definiu-se como trabalho futuro, a criação de scripts de teste que contenham casos de teste agrupados, ou seja, várias possibilidades de caminhos de execução em um único caso de teste (Figura 26 - A). Dessa forma, quando se testa uma aplicação, verifica-se se o caminho percorrido por esta (Figura 26 - B) é igual a algum dos caminhos contidos no caso de teste. Com isso, os casos de teste passam a ser não-determinísticos, diminuindo assim os efeitos causados pelo *completeness problem*.



A – Caso de teste 1 (com diversos caminhos de execução)



B – Caminho de execução da aplicação paralela

Figura 26: Casos de teste não-determinísticos

Para que isso seja possível, algumas modificações são necessárias na a geração dos casos de teste. A principal delas é a criação de regras que definam na própria criação dos casos de teste, quais caminhos são válidos para aquele modelo, o que é muito difícil de se

saber, visto que o número de caminhos possíveis que uma aplicação paralela pode executar cresce conforme seus parâmetros de execução (dados de entrada e número de processos).

Entretanto, para verificar a viabilidade dessa proposta de trabalho futuro, implementou-se um protótipo baseado em perfis de execução. Com ele, executa-se a aplicação milhares de vezes, e cria-se um *rank* probabilístico com todos os caminhos de execução possíveis que acontecerem, e através desse perfil operacional, gera-se casos de teste para todos os caminhos identificados. Através dessa abordagem, verificou-se que a taxa de identificação de "falsos *bugs*" diminuiu para pouco mais de 3%, o que não interfere no processo de identificação de falhas.

Como a abordagem utilizada nesse protótipo para a geração de casos de teste não é derivada de um modelo abstrato, e sim da execução de uma aplicação aparentemente correta, nada garante que os casos de teste gerados estejam corretos. Entretanto, os dados obtidos com o protótipo ajudam a perceber que a proposta de casos de teste agrupados é uma solução viável para a geração de casos de teste não-determinísticos, e assim, uma possível solução para a validação do processo de teste.

Capítulo 6

DIMINUIÇÃO DA INTRUSÃO

No Capítulo anterior, não foi abordada nenhuma questão referente à diminuição da intrusão do teste de software. Apenas foi apresentado o ambiente de teste e suas funcionalidades, assim como os problemas encontrados. Nesse capítulo, a estratégia utilizada para a diminuição da intrusão será apresentada. Um experimento será avaliado, visando medidas de desempenho que comparem a nossa abordagem de monitoração com uma outra ferramenta comercial de monitoração.

6.1 VISÃO GERAL

Tanto as etapas de teste quanto de depuração necessitam de algum mecanismo de monitoração, seja para controlar e analisar a execução do programa a ser testado, ou até mesmo para re-execuções controladas, em caso de depuração cíclica.

Como já foi apresentado no Capítulo 4, as funções do monitor devem influenciar o mínimo possível o comportamento da aplicação. Essa influência chama-se *probe-effect* ou intrusão, e deve ser minimizada, para que o comportamento da execução do programa não seja alterado.

A intrusão está diretamente ligada com a quantidade de dados monitorados, e é um problema que se agrava quando se trata de programas paralelos. Isso ocorre, basicamente, por ter vários processos envolvidos, exigindo mais tempo para a monitoração e também mais memória para o armazenamento desses dados coletados.

Tanto a utilização quanto os tipos de arquiteturas envolvidas na estrutura de uma ferramenta de monitoração foram tratados no Capítulo 4. Aqui, busca-se apresentar a nossa abordagem de monitoração utilizada no *engine* de teste e, principalmente, mostrar como estamos buscando a diminuição da intrusão no processo de teste.

6.2 METODOLOGIA UTILIZADA

Como já mencionado no capítulo anterior, o monitor utilizado nesse trabalho é um processo central (0 - monitor) que fica constantemente controlando e observando o comportamento dos processos envolvidos na execução da aplicação, conforme pode ser observado na Figura 27. Dessa forma, a cada evento ocorrido, este recebe o estado atingido pelo processo que executou o evento, armazenando e atualizando o estado global da aplicação. Os dados coletados durante a monitoração são dados referentes apenas às mudanças de estado ocorridas, o que já torna essa abordagem menos intrusiva.

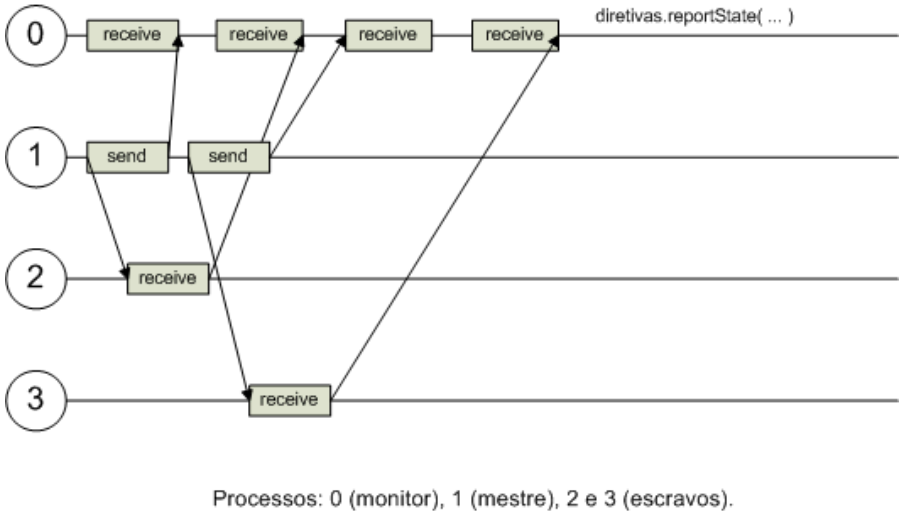


Figura 27: Comportamento do monitor do *engine*

Outra maneira de diminuir o problema da intrusão se dá na escolha do que analisar no instante do teste. Algo que envolve a estrutura dos casos de teste e scripts, assim como a instrumentação da aplicação, para que ambas sejam compatíveis com a abordagem utilizada na monitoração. Nesse caso, a diminuição da intrusão não ocorre apenas na redução dos dados de teste (estados a serem testados), mas também no que monitorar na execução da aplicação.

Visto que o *engine* de teste apresenta problemas na geração dos casos de teste, poucos testes foram feitos envolvendo as estratégias relacionadas aos casos de teste. Entretanto, a concentração maior dos testes se deu nas medidas de desempenho da técnica de monitoração utilizada no *engine*, as quais serão analisadas na próxima seção[13].

[13] Esse experimento foi formalizado em um artigo científico e será publicado nos anais do LATW 2006 (7th IEEE Latin-American TestWorkshop), conforme o Anexo B.

6.3 RESULTADOS E EXPERIMENTOS

6.3.1 AVALIAÇÃO DE DESEMPENHO

Existem diversas ferramentas de monitoração que podem servir de modelo para uma comparação de desempenho com o monitor implementado, entretanto, poucas possuem uma estrutura semelhante de dados a ser coletada. Sendo assim, uma vez que o MPE (*Multi-Processing Environment*) [75] possui essa semelhança, testes de desempenho comparativos com este foram realizados, buscando validar a diminuição da intrusão do nosso monitor.

O MPE é uma biblioteca pertencente ao pacote do MPI que fornece métodos de monitoração para análise *off-line*. Através dele, é possível coletar informações de transições de estados e sincronização entre processos. E o que é mais interessante, ele possui o conceito de estado global, visto que este é necessário para a visualização do comportamento descrito no *log* gerado. Embora o *engine* de teste não crie *log* de execução, pois possui uma abordagem de teste *on-line*, para esse experimento, algumas adaptações no *engine* foram feitas, para que este gere um *log* correspondente ao comportamento monitorado (*tracefile*). No entanto, o formato do *log* utilizado no MPE é binário, sendo lido apenas por uma ferramenta específica, a *Jumpshot* (Capítulo 4). Já em nossa abordagem, o *log* é em *ASCII*, podendo ser lido facilmente em qualquer editor de texto.

A aplicação escolhida para os experimentos continua sendo a ordenação de vetores utilizada no capítulo anterior. Nesse experimento, vinte vetores são ordenados em uma abordagem mestre-escravo. Os testes foram feitos em um *cluster* composto por quatorze máquinas, sendo todas do tipo Pentium III 500Mhz, com 256MB RAM e interconectadas por uma rede *FastEthernet* de 100Mb. Foram usados três tamanhos diferentes de vetores: 5000, 10000, e 20000 elementos do tipo “inteiro”. O número de escravos utilizados variou de 3, 5, 7, 9, e 11.

As Figuras 28, 29 e 30 apresentam os resultados obtidos a partir de uma média feita através de vinte execuções para cada item avaliado. Para facilitar a interpretação dos resultados, três gráficos diferentes foram gerados, onde as coordenadas horizontais representam o número de escravos, e as verticais, o tempo de execução obtido em segundos. Cada gráfico apresenta quatro linhas (itens): uma obtida para a versão da aplicação implementada em C/MPI; outra para a implementação em JAVA/MPI, outra para a implementação em C/MPE, e outra para a implementação em JAVA usando nosso *engine*.

A Figura 28 apresenta os resultados obtidos com vetores de 5000 elementos. Nesse caso, a versão implementada em C/MPI apresentou o melhor resultado. Também é possível notar que a intrusão nas versões da aplicação que utilizam JAVA aumenta à medida que o

número de escravos também cresce devido a necessidade de uma maior comunicação entre os processos e também a baixa computação exigida.

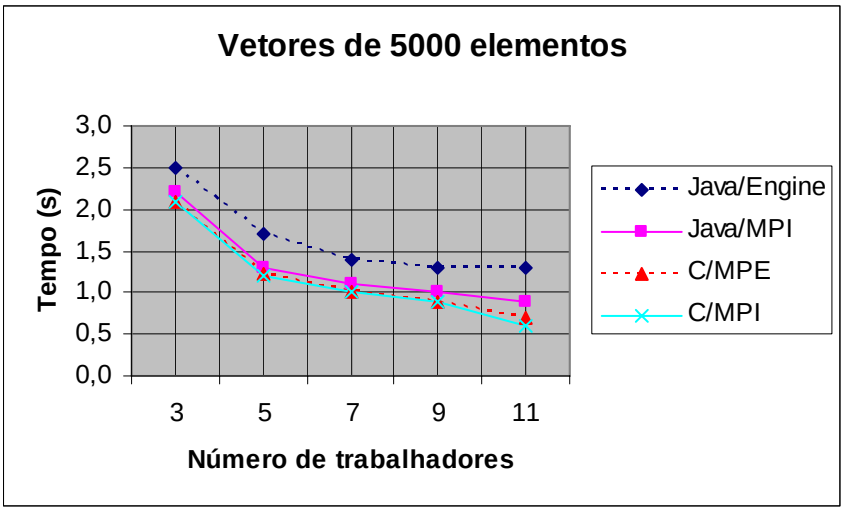


Figura 28: Intrusão medida com vetores de 5000 elementos.

De acordo com as Figuras 29 e 30, as versões em JAVA, com e sem o *engine,* apresentam uma curva decrescente constante, isso porque a quantidade de computação necessária torna-se mais significante que intrusão causada quando os dados crescem, chegando a ser quase equivalente no 11º escravo da Figura 29. No entanto, nesse mesmo ponto, a quantidade de computação por processo não é suficiente para compensar a intrusão causada pela Máquina Virtual do JAVA mais o *engine*.

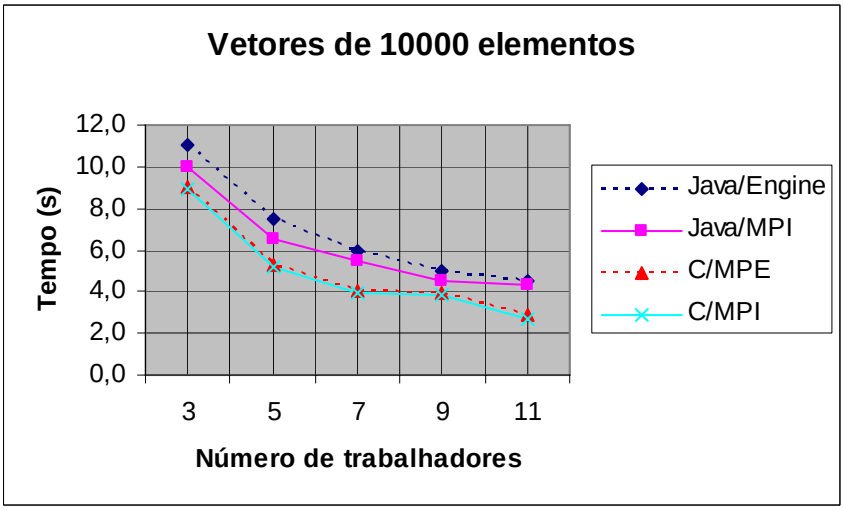


Figura 29: Intrusão medida com vetores de 10000 elementos.

E por último, é importante lembrar que, quando o tamanho do problema e o número de processos trabalhadores aumentam, a intrusão do *engine* torna-se menos significativa, tornando todos os tempos mais próximos, conforme o gráfico da Figura 30.

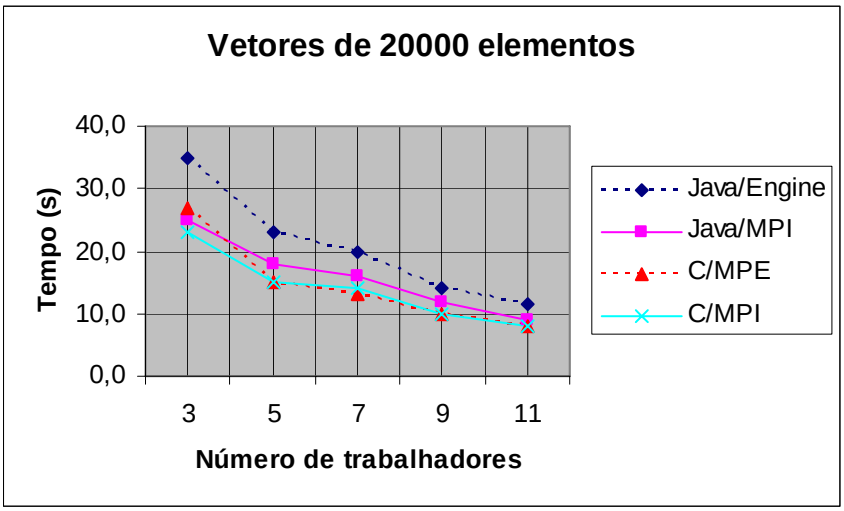


Figura 30: Intrusão medida com vetores de 20000 elementos.

Esses resultados comprovam outros trabalhos já realizados em relação ao desempenho do JAVA na tentativa de integrá-lo a uma implementação do MPI. Isso se dá através de chamadas a métodos nativos, como é o caso do mpiJAVA [76], e acaba impondo restrições de desempenho quando comparadas com implementações em C e FORTRAN [76]. No entanto, segundo [77], o mpiJAVA apresenta desempenhos mais próximos a versão nativa (MPI) quando executada com uma maior quantidade de computação e com tamanho maior de mensagem, o que pode ser percebido nas Figuras 29 e 30.

Assim conclui-se que, com os resultados obtidos, e por se tratar de uma ferramenta de monitoração implementada em JAVA, acredita-se que esta é capaz de apresentar baixa intrusão, mesmo que os gráficos demonstrem situações de desempenho inferior ao MPE. No entanto, isso pode ser visto de maneira positiva, pois o MPE quase não gera intrusão em relação ao MPI e principalmente por estas serem ferramentas que vêm sofrendo melhorias de desempenho ao longo dos anos.

Capítulo 7

CONCLUSÕES E TRABALHOS FUTUROS

A atividade de teste consiste em uma análise dinâmica do produto a ser testado, sendo relevante para a identificação e eliminação de erros que persistem constituindo um dos elementos para fornecer evidências da confiabilidade do software. O principal objetivo do teste de software é revelar a presença de falhas no produto testado, portanto, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe.

Um programa paralelo é considerado, basicamente, uma coleção de diversos processos seqüenciais que são executados simultaneamente e se comunicam de alguma forma. Conseqüentemente, as dificuldades básicas encontradas no teste e depuração de programas seqüenciais, são também evidentes para qualquer programa paralelo. Entretanto, além dessas dificuldades, os programas paralelos apresentam outros problemas relacionados com o aumento da complexidade, a quantidade de dados depurados e os efeitos anômalos adicionais, os quais muitas vezes acabam gerando uma intrusão excessiva na monitoração da aplicação em teste.

Nessa dissertação, foi proposta uma estratégia para diminuir a intrusão do teste de software em programas paralelos não-determinísticos. Busca-se com essa estratégia, validar as abordagens de monitoração assim como as funcionalidades do ambiente de teste implementado.

Conforme os resultados apresentados nos Capítulos 5 e 6, a estratégia proposta obteve resultados positivos, tendo a diminuição da intrusão validada perante as comparações de desempenho com o MPE, além da validação das funções de monitoração e atingibilidade de estados durante os experimentos realizados. Entretanto, os problemas encontrados com os "falsos *bugs*" impediram que nesse trabalho fosse validada também, a capacidade do *engine* de teste de apresentar o resultado do teste, o que nos incentivou a buscar novos experimentos para identificar possíveis soluções para esse problema.

Essas soluções representam no escopo do projeto, alterações nas etapas de geração de casos de teste e scripts, sendo inviáveis de implementá-las em tempo hábil, dado a etapa final do projeto em que nos encontramos. Com isso, definiu-se como trabalho futuro, a criação de

scripts de teste que contenham casos de teste agrupados. Espera-se que dessa forma seja possível dar continuidade a todos os esforços dedicados nessa dissertação.

Referências

[1] Oliveira. F.M., Copstein. B., Reginato. L. R. C. "***STAGE: an Integrated Environment for Statistical Test Script Generation***". V Workshop de Testes e Tolerância a Falhas, pp. (77-88), UFRGS, Gramado, Maio, 2004.

[2] Hwang. K., Xu. Z. "***Scalable Parallel Computing: Technology, Architecture, Programming***". McGraw-Hill, San Francisco, Janeiro, 1998, 802p.

[3] Foster. I.T. "***Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering***". Addison-Wesley, Boston, Janeiro, 1995, 430p.

[4] Wilkinson. B., Allen. M. "***Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers***". Prentice-Hall, New York, Agosto, 1998, 431p.

[5] De Rose. C. A. F., Navaux. P. O. A. "***Arquiteturas Paralelas***". Sagra-Luzzato, Porto Alegre, Maio, 2003, 115p.

[6] Alsberg. P. A., Mills. C. R. "***The Structure of the ILLIAC IV Operating System***". ACM Symposium on Operating Systems Principles, pp. (92-96), ACM Press, Princeton, Junho, 1969.

[7] "***Top500 Computing Sites***". Disponível em: http://www.top500.org, Último acesso: Dezembro, 2005.

[8] Hennessy. J.L., Patterson. D.A. "***Computer Architecture: A Quantitative Approach***". Morgan Kaufmann, San Francisco, Janeiro, 1996, 760p.

[9] Amza. C., et al. "***Trademarks: Shared memory computing on networks of workstations".*** IEEE Transactions on Computer, Vol. 29**(**2**)**, pp. (18-28), IEEE Computer Society Press, New Jersey, Fevereiro, 1996.

[10] Flynn. M. J. "***Some Computer Organizations and their Effectiveness***". IEEE Transactions on Computers, Vol. 24 **(**9**)**, pp. (948-960), IEEE Computer Society Press, New Jersey, Setembro, 1972.

[11] Buyya. R. "***High Performance Cluster Computing - Architectures and Systems***". Prentice-Hall, New Jersey, Maio, 1999, 849p.

[12] Buyya. R. "***High Performance Cluster Computing - Programming and Applications***". Prentice-Hall, New Jersey, Junho, 1999, 664p.

[13] Brune. M., Gehring. J., Reinefeld. A. "***Linking Message-Passing Environments***". in: Buyya, R. "High Performance Cluster Computing - Programming and Applications". Prentice-Hall, New Jersey, Junho, 1999, 664p.

[14] Leinberger. W., Kumar. V. "***Information Power Grid: The New Frontier in Parallel Computing***". IEEE Concurrency, Vol. 7**(**4**)**, pp. (75-84), IEEE Computer Society Press, New York, Dezembro, 1999.

[15] Grimshaw. A., Ferrari. A., Lindahl. G., Holcomb. K. "***Metasystems***". Communications of the ACM, Vol. 41**(**11**)**, pp. (46-55), ACM Press, New York, Novembro, 1998.

[16] Foster. I., Kesselman. C. "***The Globus Project: A Status Report"***. Special Issue on Metacomputing, Vol. 15**(**5-6**),** pp. (607-621), Elsevier Science Publishers, Amsterdam, Março, 1999.

[17] El-Rewini. H., Lewis. T., Shriver. B. "***Parallel and Distributed Systems - From Theory to Practice***". IEEE Parallel and Distributed Technology, Vol. 1**(**3**)**, pp. (7-11), IEEE Computer Society Press, New York, Agosto, 1993.

[18] Wismüller. R. "***State Based Visualization of PVM Applications***". Parallel Virtual Machine, Lecture Notes in Computer Science, Vol. 1156, pp. (91-99), Springer Verlag, Munich, Outubro, 1996.

[19] Geist. G.A., Sunderam. V. S. "***Network-based Concurrent Computing on the PVM System***". Concurrency - Practice & Experience, Vol. 4**(**4**)**, pp. (293-311), John Wiley and Sons Ltd, Chichester, Junho, 1992.

[20] "***MPI: A Message-Passing Interface Standard – Version 1.1***". Disponível em: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html, Último acesso: Dezembro, 2005.

[21] Boehm. B.W. "***Software Engineering***". IEEE Transactions on Computers, Vol. 25**(**12**)**, pp. (1226-1241), IEEE Computer Society Press, New Jersey, Dezembro, 1976.

[22] Boehm. B.W., Port. D. "***Educating Software Engineering Students to Manage Risk***". 23rd International Conference on Software Engineering, pp. (591-600), IEEE Computer Society Press, Washington, Maio, 2001.

[23] Paulk. M. C., et al. "***Capability Maturity Model, Version 1.1***". IEEE Software, Vol. 10**(**4**)**, pp. (18-27), IEEE Computer Society Press, Los Alamitos, Julho, 1993.

[24] Paulk. M. C., et al. "***The Capability Maturity Model: Guideline for Improving the Software Process***". SEI – Series in Software Engineering, Addison-Wesley Publishing, Boston, Janeiro, 1995, 441p.

[25] Herbsleb. J., et al. "***Software Quality and the Capability Maturity Model***". Communication of the ACM, Vol. 40**(**6**)**, pp. (30-40), ACM Press, New York, Setembro, 1997.

[26] "***Capability Maturity Model for Software***". Disponível em: http://www.sei.cmu.edu/cmm/, Último acesso: Dezembro, 2005.

[27] "***IEEE Standard Glossary of Software Engineering Terminology***". Disponível em: http://standards.ieee.org/catalog/, Último acesso: Dezembro, 2005.

[28] Avizienis, A. "***Toward Systematic Design of Fault-Tolerant Systems***". IEEE Computer, Vol. 30**(**4**)**, pp. (51-58), IEEE Computer Society Press, Los Alamitos, Abril, 1997.

[29] Pradhan. D. K. "***Fault-Tolerant Computer System Design".*** Prentice-Hall Advanced Computing and Telecommunications Series, Prentice-Hall Inc., Saddle River, Novembro, 1996, 550p.

[30] Wasserman. H., Blum. M. "***Software Reliability via Run-Time Result-Checking***". Journal of the ACM, Vol. 44**(**6**)**, pp. (826-849), ACM Press, New York, Dezembro, 1997.

[31] Sayre. K. "***Improved techniques for software testing based on Markov chain usage models***". Ph.D Thesis, 118p., University of Tennessee, Dezembro, 1999.

[32] Trammell. C. "***Quantifying the reliability of software: statistical testing based on a usage model***". IEEE Software Engineering Standards Symposium, Vol. 54**(8),** pp. (208-218), IEEE Computer Society Press, Washington, Maio, 1995.

[33] Norris. J. R. "***Markov Chains***". Cambridge University Press, New York, Maio, 1998, 237p.

[34] Stewart. W. J. "***Introduction to the numerical solution of Markov Chains***". USA Princeton University Press, New Jersey, Novembro, 1994, 539p.

[35] Plateau. B., Atif. K. "***Stochastic automata networks for modeling parallel systems***". IEEE Transactions on Software Engineering, Vol. 17(10), pp. (1093-1108), IEEE Computer Society Press, Dezembro, 1991.

[36] Fernandes. P., Plateau. B., Stewart. W. J. "***Efficient descriptor-vector multiplication in stochastic automata networks***". Journal of the ACM, Vol. 45(3), pp. (381-414), ACM Press, Janeiro, 1998.

[37] Whittaker. J.A., Thomason. M.G. "***A Markov Chain Model for Statistical Software Testing***". IEEE Transactions on Software Engineering, Vol. 20(10), pp. (812-824), IEEE Computer Society Press, Piscataway, Setembro, 1994.

[38] Farina. A.G., Fernandes. P., Oliveira. F.M. ***"Representing Software Usage Models with Stochastic Automata Networks"***. International Conference on Software Engineering and Knowledge Engineering, Vol. 27, ACM Press, New York, Março, 2002.

[39] Bertolini. C., et al. "***Test Case Generation Using Stochastic Automata Networks: Quantitative Analysis"***. 2th International Conference on Software Engineering and Formal Methods, pp. (251-260), IEEE Computer Society, Beijing, Setembro, 2004.

[40] Baldo. L., et al "***Performance Models for Master/Slave Parallel Programs"***. 1th Internationl Workshop on Practical Applications of Stochastic Modeling, Vol.28(4), pp. (101-121), Elsevier Science Press, London, Maio, 2004.

[41] Kranzlmüller. D., Volkert. J. "***Debugging Point-to-Point Communication in MPI and PVM***". 5th European PVM/MPI Users Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Vol.1497, pp. (265-272), Springer Verlag, London, Agosto, 1998.

[42] Kranzlmüller. D., Grabner. S., Volkert. J. "***Monitoring Strategies for Hypercube Systems***". 4th Euromicro Workshop on Parallel and Distributed Processing, p. 486, IEEE Computer Society Press, Washington, Abril, 1996.

[43] Kranzlmüller. D., Grabner. S., Volkert. J. ***"Message Passing Visualization with ATEMPT"***. Advances in Parallel Computing, Vol. 11, pp. (653-656), Elsevier Science Press, North Holland, 1996.

[44] Lourenco. J., et al. ***"An Integrated Testing and Debugging Environment for Parallel and Distributed Programs"***. 23rd EUROMICRO Conference, pp. (291-298), IEEE Computer Society Press, Budapest, Junho, 1997.

[45] Paul. J. "***Software Engineering - General Testing and Debugging Guidelines***". Disponível em: http://www.jodypaul.com/SWE/TD/TestDebug.html, Último acesso: Dezembro, 2005.

[46] Walton. G. H., Poore. J. H., Trammell. C. J. "***Statistical Testing of Software Based on a Usage Model***". Software - Practice and Experience, Vol. 25(1), pp. (97-108), John Wiley & Sons, Inc., New York, Agosto, 1995.

[47] Choi. J., Miller. B. P., Netzer. R. H. B. "***Techniques for Debugging Parallel Programs with Flowback Analysis***". ACM Transactions on Programming Languages and Systems, Vol. 13(4), pp. (491-530), ACM Press, New York, Agosto, 1991.

[48] Feldman. S. I., Brown. C. B. "***Igor: A System for Program Debugging via Reversible Execution***". Workshop on Parallel and Distributed Debugging, Vol. 24(1), pp. (112-123), ACM Press, New York, Setembro, 1989.

[49] Weiser. M. "***Program Slicing***". IEEE International Conference on Software Engineering, Vol. 10(4), pp. (439-449), IEEE Computer Society Press, Piscataway, Outubro, 1981.

[50] LeDoux. C. H., Parker. D. S. Jr. "***Saving Traces for Ada Debugging***". International Conference on ADA, pp. (97-108), Cambridge University Press, New York, Março, 1985.

[51] Rosenberg. J.B. ***"How Debuggers Work: Algorithms, Data Structures, and Architecture"***. John Wiley & Sons, Inc., New York, Outubro, 1996, 256p.

[52] Baecker. R., DiGiano. Ch., Marcus. A. "***Software Visualization for Debugging***". Communications of the ACM, Vol. 40(4), pp. (44-54), ACM Press, New York, Março, 1997.

[53] Grabner. S., Volkert. J. "***Debugging Distributed Memory Programs Using Communication Graph Manipulation***". International Symposium on High Performance Computing Systems, Springer Verlag, Canada, Vol. 35(7), pp. (51- 59), New York, Outubro, 1996.

[54] Netzer. R. H. B., Miller. B. P. ***"Optimal Tracing and Replay for Debugging Message-Passing Parallel Program"***. ACM/IEEE Conference on Supercomputing, Vol.20(3), pp. (502-511), IEEE Computer Society Press, Los Alamitos, Abril, 1992.

[55] Leu. E., Schiper. A., Zramdini. A. "***Execution Replay on Distributed Memory Architectures***". 2th IEEE Symposium on Parallel and Distributed Processing, pp. (106-112), IEEE Computer Society Press, Dezembro, 1990.

[56] Tai. K. C., Carver. R. H. ***"Testing Distributed Programs"***. in: Zomaya. A.Y., "Parallel and Distributed Computing Handbook", Mcgraw-Hill Computer Engineering Series, New York, Agosto, 1996, 1232p.

[57] Oberhuber. M. ***"Elimination of Nondeterminacy for Testing and Debugging Parallel Programs"***. International Workshop on Automated and Algorithmic Debugging, pp. (315-316), Saint Malo, Setembro, 1995.

[58] Hondroudakis. A. "***Performance Analysis Tools for Parallel Programs".*** Technical Report, Edinburgh Parallel Computing Centre, The University of Edinburgh, Julho, 1995.

[59] Hood. R. "***The p2d2 Project: Building a Portable Distributed Debugger"***. Symposium on Parallel and Distributed Tools, pp. (127-136), ACM Press, New York, Março, 1996.

[60] Cunha. J. C., et al. ***"A Framework to Support Parallel and Distributed Debugging"***. International Conference and Exhibition on High-Performance Computing and Networking, Vol. 14(1), pp. (708-717), Springer Verlag, London, Abril, 1998.

[61] Clemencon, C., Fritscher, J., Rühl, R. "***Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool"***. High Performance Computing Symposium, pp. (393-404), Montreal, Julho, 1995.

[62] Appelbe. W. F., McDowell. C. E. ***"Integrating Tools for Debugging and Developing Multitasking Programs"***. ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Vol. **24**(1), pp. (78-88), ACM Press, New York, Julho, 1989.

[63] Krawczyk. H., Wiszniewski. B. "***Interactive Testing Tool for Parallel Programs"***. Software Engineering for Parallel and Distributed Systems, Chapman Hall, London, Março, 1996, 380p.

[64] Kacsuk. P., et al. "***A Graphical Development and Debugging Environment for Parallel Programs"***. Journal of Parallel Computing, Distributed and Parallel Systems: Environments and Tools, Vol. **22**(13), pp. (1747-1770), Elsevier Science Press, New York, Outubro, 1997.

[65] Kranzlmuller. D., Grabner. S., Volkert. J. "***Debugging with the MAD environment"***. Parallel Computing, Special double issue on environment and tools for parallel scientific computing, Vol. 23(1-2), pp. (199-217), Elsevier Science Press, Amsterdam, Maio, 1997.

[66] Vautherin. J. ***"Parallel Systems Specifications with Coloured Petri Nets and Algebraic Abstraction Data Types"***. 7th European Workshop on Applications and Theory of Petri Nets, pp. (293-308), Springer Verlag, Berlim, Setembro, 1996.

[67] Beguelin. A., et al. ***"Visualization and Debugging in a Heterogeneous Environment"***. IEEE Computer, Vol. **26**(6), pp. (88-95), IEEE Computer Society Press, Los Alamitos, Março, 1993.

[68] Lamport. L. "***Time, Clocks, and the Ordering of Events in a Distributed System***". Communications of the ACM, Vol. **21**(7), pp. (558-565), ACM Press, New York, Março, 1978.

[69] Browne. J. C., et al. "***Visual Programming and Debugging for Parallel Computing"***. IEEE Parallel & Distributed Technology: System & Technology, Vol. **3**(1), pp. (75-83), IEEE Computer Society Press, Los Alamitos, Outubro, 1995.

[70] Heath. M. T. ***"Recent Developments and Case Studies in Performance Visualization using ParaGraph"***. Workshop on Performance Measurement and Visualization of Parallel Systems, pp. (175-200), Elsevier Science Press, Amsterdam, Setembro, 1993.

[71] Yan. J. C., Jin. H. H., Schmidt. M. A. "***Performance Data Gathering and Representation from Fixed-Size Statistical Data"***. *Technical Report,* NAS Systems Division, NASA Ames Research Center, California, 1998.

[72] Geist. G. A., Kohl. J., Papadopoulos. P. "***Visualization, Debugging, and Performance in PVM"***. Debugging and Performance Tuning for Parallel Computing Systems, IEEE Computer Society Press, Los Alamitos, Outubro, 1996.

[73] Herrarte. V., Lusk. E. "***Studying parallel program behavior with Upshot"***. *Technical Report*, Argonne National Laboratory, Argone, Abril, 1991.

[74] Nagel. W. E., et al. ***"VAMPIR: Visualization and Analysis of MPI Resources"***. Supercomputer, Vol. 12(1), pp. (69-80), Elsevier Science Press, New York, Agosto, 1996.

[75] Chan. A., Gropp. W., Lusk. E. "***Scalable Log Files for Parallel Program Trace Data***".

Technical Report, Argonne National Laboratory, Argonne, Agosto, 2000.

[76] Baker. M., et al. "***MPIJAVA: An Object-Oriented JAVA Interface to MPI***". 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, Vol. 1586, pp. (748-762), Springer Verlag, London, Junho, 1999.

[77] Judd. G., et al. "***Design issues for efficient implementation of MPI in Java***". Proceedings of the ACM Conference on Java Grande, pp. (58-65), ACM Press, New York, Setembro, 1999.

Anexo A

ARTIGO - CLEI 2005

Trabalho aprovado e apresentado na Conferência Latino Americana de Informática (CLEI 2005) em outubro de 2005, na cidade de Cali, Colômbia.

Estratégia para desenvolvimento de programas paralelos visando diminuir a intrusão causada por teste de software*

Leonardo Albernaz Amaral, Eduardo Augusto Bezerra

Hewlett-Packard/PUCRS – Centro de Pesquisa em Teste de Software (CPTS)
Faculdade de Informática (FACIN/PPGCC), PUCRS
Av. Ipiranga, 6681 – Prédio 30, 90619-900, Porto Alegre, RS, Brasil

{lamaral, eduardob}@inf.pucrs.br

Abstract

This work introduces a strategy to be used as part of a parallel programming methodology, aiming the reduction of the intrusion caused by software testing activities. In the proposed strategy, a discrete formalism (SAN) is used for the behavioural model representation of parallel applications. Test cases are created from this behavioural model to stimulate parallel programs, seeking out for inter-process communication errors. An important contribution of the proposed strategy is the generation and use of SAN-based test cases to reach coverage criteria, according to probabilities obtained from behavioural rates defined in the application model. The main idea behind the strategy is to come up with more deterministic test coverage schemes, mitigating the problem related to the large number of possible execution paths present in non-deterministic parallel programs.

Keywords: *parallel program testing and debugging, discrete formalism models, Stochastic Automata Networks*.

Resumo

Neste trabalho é apresentada uma estratégia para desenvolvimento de programas paralelos visando diminuir a intrusão causada pelas etapas de teste de software. A proposta baseia-se basicamente na idéia de utilizar um formalismo discreto (SAN) para a representação do modelo comportamental da aplicação e dessa forma criar casos de teste que exercitem a aplicação paralela na busca por falhas de comunicação entre processos. Uma contribuição importante dessa proposta é a utilização de casos de teste criados através de modelos SAN, o que possibilita critérios de cobertura baseados nas probabilidades obtidas a partir das taxas comportamentais determinadas no modelo da aplicação. Busca-se com essa estratégia, conseguir critérios de cobertura de teste mais determinísticos, diminuindo assim o problema do número excessivo de caminhos possíveis de execução geralmente encontrados em programas paralelos não-determinísticos.

Palavras chaves: *teste e depuração de programas paralelos, modelos de formalismo discreto, Redes de Autômatos Estocásticos*.

1. INTRODUÇÃO

O processamento paralelo pode ser considerado um método computacional onde se dividem grandes problemas em problemas menores que são processados simultaneamente por diferentes processadores, buscando-se alto desempenho na solução destes. A crescente aceitação do processamento paralelo deve-se principalmente ao aperfeiçoamento das redes de computadores, ao surgimento de *clusters* computacionais e também a computação distribuída [15].

Um *cluster* é basicamente uma coleção de máquinas distribuídas através de uma rede de alta velocidade, que serve como meio de comunicação entre essas máquinas. Seu potencial pode ser explorado adequadamente com a utilização do paradigma da troca de mensagens. Existem algumas opções disponíveis para dar suporte ao desenvolvimento de aplicações paralelas baseadas nesse paradigma como, por exemplo, o MPI (*Message Passing Interface*), que é implementado por meio de uma biblioteca de funções que provê suporte à comunicação entre processos.

Os *clusters* oferecem um enorme poder computacional e são usados para solucionar grandes desafios disseminados em diversas áreas estratégicas do conhecimento humano como: simulação e modelos de predição de desempenho (previsão do tempo, bioinformática, oceanografia), exploração de recursos de energia (explorações sísmicas e projetos de reservatórios), pesquisas médicas (tomografia computadorizada e estudos da engenharia

* Este trabalho foi desenvolvido em colaboração com a HP Brasil R&D.

genética), pesquisas militares (projetos de armas e simulações de ataques) efeitos visuais (filmes e animações), entre outros. O *software* para essa categoria de computadores deve ser desenvolvido de forma a fornecer resultados precisos e confiáveis, sendo o teste uma etapa bastante importante para o aumento dessa confiabilidade necessária.

Para o teste de *software* funcional, um programa seqüencial ou paralelo pode ser visto como um conjunto de estados e transições entre esses estados, as quais determinam o comportamento do sistema durante a sua execução. Para tal estratégia de teste, o comportamento do sistema pode ser abstraído através de formalismos para representação de modelos de uso e verificado através da consistência do comportamento observado durante a execução do programa, com o comportamento esperado, geralmente baseado na especificação do mesmo [51]. Em programas paralelos, as transições entre estados geralmente apresentam-se na forma de eventos de comunicação entre processos.

A descrição de aplicações por meio de modelos de uso é uma alternativa que vêm sendo utilizada pela comunidade científica, como uma estratégia para caracterizar o comportamento de um sistema, podendo dar mais detalhes às especificações do mesmo, principalmente quando associados a métodos estatísticos [52, 53, 6, 4].

Nesse trabalho é apresentada uma estratégia para desenvolvimento de programas paralelos visando diminuir a intrusão causada por teste de *software*. Essa proposta baseia-se em utilizar modelos de formalismo discreto (*SAN*) para a representação do modelo comportamental da aplicação paralela, e com isso mapear o modelo com o código da aplicação, determinando assim, o mapeamento dos eventos do modelo *SAN* com os eventos da aplicação. Dessa forma, através das estimativas probabilísticas de uso inferidas no modelo, cria-se casos de teste e *scripts* de teste funcional para a aplicação a ser testada e se define critérios de cobertura que permitam testar os programas paralelos utilizando-se além de técnicas de teste de *software*, algumas abordagens de depuração de programas paralelos.

As próximas Seções desse trabalho estão organizadas da seguinte maneira. As Seções 2 e 3 descrevem sucintamente os conceitos que serão utilizados e os principais trabalhos relacionados às ferramentas para criação, manipulação e análise de programas paralelos. Na Seção 4 é feito um detalhamento da proposta e uma discussão do andamento do trabalho. E por último, a Seção 5 apresenta algumas conclusões e trabalhos futuros.

2. TESTE DE SOFTWARE

Em conseqüência da crescente utilização de sistemas computacionais em praticamente todas as áreas do conhecimento humano, nas últimas décadas a Engenharia de *Software* evoluiu significativamente, procurando estabelecer técnicas, critérios, métodos e ferramentas para a produção de *software*. Tudo isso impulsionado por uma crescente demanda por qualidade e produtividade, tanto do ponto de vista do processo de produção, quanto dos produtos gerados.

Dentro desse contexto, a atividade de teste de *software* consiste basicamente em uma análise dinâmica do produto a ser testado, sendo considerada uma atividade relevante para a identificação e eliminação de falhas existentes em um sistema. Erros geralmente ocorrem devido a algum tipo de falha, seja esta humana ou física. Sistemas em estado errôneo são considerados sistemas defeituosos, visto que o processamento posterior a esse estado possivelmente cause defeitos operacionais, fazendo com que o sistema não se comporte conforme o esperado pela sua especificação [44, 7, 8]. O conjunto de informações oriundas da atividade de teste é significativo para as atividades de depuração, manutenção e estimativa de confiabilidade de *software* [19, 49].

Uma questão importante da atividade de teste, independentemente da fase, é a avaliação da qualidade de um determinado conjunto de casos de teste, visto que dependendo da aplicação, pode ser impraticável utilizar todo o domínio de dados de entrada para avaliar os aspectos funcionais e operacionais de um produto em teste. Assim, o objetivo do teste é utilizar casos de teste que tenham alta probabilidade de encontrar a maioria dos erros com um mínimo de tempo e esforço. Portanto, um teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe.

Basicamente existem três estratégias que podem ser aplicadas à fase de teste: a funcional *(Black Box*), a estrutural (G*lass Box*) e a baseada em métodos estatísticos. Na técnica funcional, os critérios e requisitos de teste são estabelecidos a partir de uma função de especificação do *software*, onde o objetivo é determinar se o programa satisfaz aos requisitos funcionais e não-funcionais que foram especificados [43]. Na técnica estrutural, os critérios e requisitos são derivados essencialmente a partir das características de uma particular implementação em teste, o que requer a inspeção do código fonte e a seleção de casos de teste que exercitem partes do código e não de sua especificação [9]. E na técnica baseada em métodos estatísticos, os critérios e requisitos de teste permitem o uso de inferências estatísticas para computar aspectos probabilísticos do processo de teste, tais como confiabilidade, tempo médio para falha e tempo médio entre falhas [12]. Observa-se também o estabelecimento de critérios de geração de casos de teste baseados em Máquinas de Estados Finito [16], Cadeias de Markov [53] e mais recentemente baseados em Redes de Autômatos Estocásticos [6].

2.1 Formalismo para Teste Estatístico

O teste estatístico com o passar dos anos passou a despertar grande interesse, pois viabiliza a superação de alguns pontos fracos de outras estratégias de teste. Ele geralmente está baseado em modelos de uso, os quais podem descrever possíveis comportamentos de um determinado *software*.

Os modelos de uso geralmente são representados por algum tipo de formalismo. Esses formalismos na maioria das vezes são baseados em grafos de estados com probabilidades de transição. O primeiro formalismo utilizado no teste estatístico foi Cadeias de *Markov*. Entretanto, existem estudos recentes em torno de um outro formalismo chamado Redes de Autômatos Estocásticos - *SAN* (*Stochastic Automata Networks*).

Na medida em que os modelos markovianos aumentam de tamanho e complexidade de seus componentes, as Cadeias de *Markov* apresentam limitações na capacidade de comportar tal crescimento. Nesse ponto, as Redes de Autômatos Estocásticos oferecem a capacidade necessária para tal representação.

Basicamente uma *SAN* é considerada um formalismo capaz de modelar um sistema em vários subsistemas, ou seja, um sistema composto de módulos quase independentes. Cada subsistema é representado por um autômato estocástico e por transições entre os estados deste autômato. Cada autômato é representado por um determinado número de estados, juntamente com regras ou funções de probabilidade que regem os movimentos de um estado para outro do autômato. O estado local de um autômato no tempo *t* é o estado que este autômato ocupa no tempo *t*. Já o estado global da rede de autômatos estocásticos é dado pelo estado local que cada um dos autômatos constituintes ocupa no tempo *t*. Os eventos em uma *SAN* podem ser eventos locais ou sincronizantes, e permitem mudar o estado global de uma *SAN*. Os eventos locais alteram o estado de apenas um autômato. Já os eventos sincronizantes, alteram simultaneamente mais de um estado local, ou seja, promovem a alteração de estados em mais de um autômato ao mesmo tempo.

O uso de Redes de Autômatos Estocásticos na representação de modelos de uso é exemplificado na Figura 1, onde é abstraído o comportamento de usuários na autenticação de um sistema. Sua estrutura possui os seguintes componentes:

- Autômatos (2): {*Login Automaton*, *Password Automaton*};
- Estados (6): {*Start*, *Menu*, *Password*} e {*PNotOk*, *Waiting*, *POK*};
- Eventos (5): {*ST*, *QT*, *S*} eventos sincronizantes e {*g*, *f*} eventos locais;

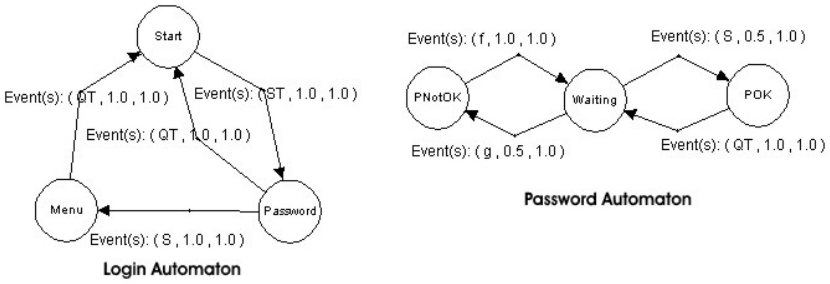


Fig. 1: Modelo *SAN* do sistema de *login*

O comportamento do sistema pode ser observado através de dois casos de teste para esse modelo, conforme a Figura 2. Nesses casos de teste são apresentados passo-a-passo os estados globais da rede e o evento que gerou a transição de estados. É possível notar que sempre que ocorre um evento sincronizante, os estados dos autômatos envolvidos nesse evento são alterados simultaneamente. Já quando ocorrem eventos locais, apenas estados locais ao evento são alterados.

Os casos de teste da Figura 2 foram criados em um ambiente integrado para geração de casos de teste e *scripts* para teste estatístico de *software* – STAGE [42]. É importante ressaltar que o ambiente STAGE foi desenvolvido no CPTS/PUCRS, em um projeto em colaboração com a HP Brasil. O formalismo utilizado na criação dos casos de teste foi *SAN,* e sofreu uma série de adaptações no modelo original. A principal contribuição desse sistema foi o uso desse formalismo para a representação dos modelos de uso e um modelo intermediário (*ISEM* – Modelo de Estado de Interface) para mapear a abstração do modelo de uso com a interface dos componentes do sistema. Dessa forma, a utilização de *SAN* permitiu uma representação modular de sistemas com comportamento complexo não-deterministico, minimizando a explosão de espaço de estados apresentado em Cadeias de *Markov*.

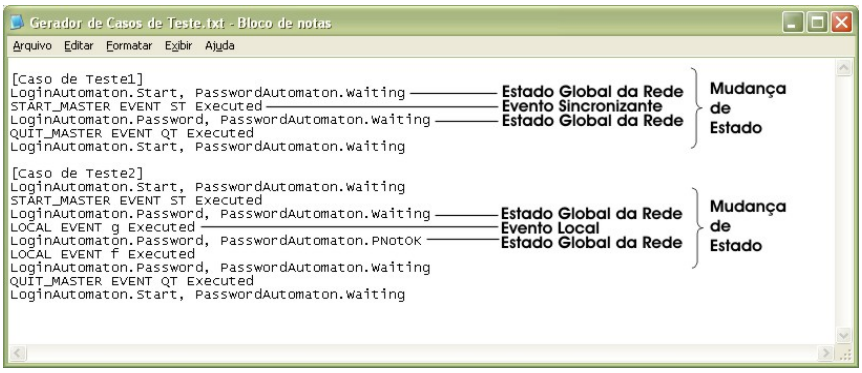


Fig. 2: Casos de teste do modelo *SAN*

No presente trabalho, busca-se estender as funcionalidades do STAGE para comportar a complexidade de programas paralelos, e com isso gerar automaticamente casos de teste e *scripts* para teste funcional de programas paralelos.

Além do trabalho apresentado em [42], outros trabalhos demonstram a utilização de modelos de uso e métodos estatísticos em etapas de teste de *software*, seja em programas seqüenciais ou em programas paralelos. Em [1] métodos estatísticos foram utilizados para analisar o comportamento de sistemas. A idéia do autor foi utilizar esses métodos para caracterizar as execuções de programas paralelos em modelos comportamentais e assim poder visualizar conjuntos específicos de dados inferidos estatisticamente através do modelo. Em [14], diversos modelos *SAN* foram criados e comparados com modelos de Cadeias de *Markov* em termos de números de estados, escalabilidade e capacidade de leitura desses modelos. Nessa comparação, *SAN* apresentou-se superior. Em [6], casos de teste foram gerados automaticamente através de modelos *SAN*. Já em [7], *SAN* é utilizada na construção de modelos de desempenho aplicados a programas paralelos, representando a comunicação entre processos paralelos com comportamentos síncronos e assíncronos.

Como a proposta desse trabalho está baseada na utilização de modelos de formalismo discreto para a representação comportamental de aplicações paralelas, fundamenta-se a utilização de Redes de Autômatos Estocásticos para essa abstração, devido a sua capacidade e flexibilidade para comportar a explosão de espaço de estados ocorridos na execução simultânea de processos paralelos. Entretanto, as etapas de teste tornam-se um pouco mais complicadas do que em programas seqüenciais, principalmente devido ao grande número de comunicação e sincronismo necessários. Dessa forma, torna-se indispensável a utilização de técnicas de depuração para a manipulação desses eventos de comunicação [30, 35, 32, 33].

Na próxima Seção, alguns conceitos relacionados a teste e depuração serão abordados, assim como os principais trabalhos que utilizam essa combinação de estratégias.

3. TESTE E DEPURAÇÃO DE PROGRAMAS PARALELOS

A princípio, um programa paralelo é uma coleção de diversos processos seqüenciais que são executados simultaneamente e se comunicam de alguma forma. Conseqüentemente, as dificuldades básicas encontradas em depuradores seqüenciais, são também evidentes para qualquer depurador paralelo [45]. Entretanto, além dessas dificuldades, os programas paralelos apresentam outros problemas. Em [30], o autor apresenta três justificativas para esses problemas: o aumento da complexidade dos programas paralelos, a quantidade de dados depurados e os efeitos anômalos adicionais.

O aumento da complexidade dos programas paralelos refere-se basicamente ao fato de que em programas paralelos se têm vários processos simultâneos, o que dificulta a estratégia tradicional utilizada em programas seqüenciais [18]. A segunda razão relaciona-se às dificuldades de se observar o que realmente interessa na depuração, evitando o acúmulo desnecessário de informações coletadas [46]. E a última razão diz respeito às dificuldades que ocorrem devido à concorrência e ao sincronismo entre os processos, o que não acontece nos programas seqüenciais [30]. Algumas dessas dificuldades são: *race-conditions* e *probe effects*. E outras se referem ao não-determinísmo e suas implicações nos programas paralelos como *irreproducibility effect* e *completeness problem*.

Um comportamento não-deterministico é caracterizado por violar o determinismo de execução, ou seja, múltiplas execuções de um mesmo programa resultando em diferentes seqüências de eventos sincronizantes e podendo produzir diferentes caminhos de execução [38]. Um exemplo de função de comunicação que introduz o não-determinismo é a função de recebimento de mensagens *MPI_Recv (buffer, any_source)* da biblioteca de funções do *MPI,* onde o parâmetro *any_source* permite o recebimento aleatório de várias mensagens. Dessa forma

não é possível determinar qual das mensagens será aceita primeiro. Isso vai depender da ordem de chegada dessas mensagens, o que caracteriza uma situação não-determinística, e com condição de corrida (*race-condition*) [39].

Existem dois problemas principais causados pelo não-determinismo, o *irreproducibility effect* e o *completeness problem*. O primeiro é caracterizado como sendo a incapacidade de realizar depuração cíclica (*cyclic debugging)*, ou seja, a incapacidade de reprodução de um erro encontrado na fase de teste, visto que o programa está sujeito a gerar a cada execução subseqüente, seqüências diferentes de eventos sincronizantes, ou diferentes caminhos de execução. Já o *completeness problem* está diretamente ligado a critérios de cobertura de teste*,* sendo um problema encontrado quando se tenta testar o comportamento de um programa paralelo em todos os caminhos possíveis de execução, o que dependendo do número de caminhos a serem testados pode se tornar impraticável.

Segundo [36], a solução para o *irreproducibility effect,* é a criação de um mecanismo que crie re-execuções equivalentes a uma execução anterior "observada". Esse mecanismo foi chamado de *record&replay,* e é composto por duas etapas. A primeira etapa é a fase de coleta (*record fase*), onde a ordem das mensagens são armazenadas em um histórico de execução (*tracefile*), o qual mantem informações sobre todos os eventos não-determinísticos. E a segunda etapa é a fase de re-execução (*replay*), onde os dados armazenados são utilizados para criar re-execuções equivalentes à execução "observada".

Para o tratamento do *completeness problem,* algumas técnicas também foram criadas como *controlled execution*, *event manipulation* e *artificial replay*. *Controlled execution* é uma abordagem proposta por [48], que oferece um método para a realização de teste automático. Baseada em uma técnica que descreve o comportamento desejado das comunicações entre os processos, o programa paralelo é executado em um comportamento forçado, evitando assim situações de condição de corrida. Uma outra abordagem dessa técnica foi proposta por [41], onde padrões de controle são aplicados para estabelecer as ordens entre os eventos de comunicação. Tais padrões são regras dinâmicas que definem a ordem de interação entre os processos.

Outros trabalhos tratam o *completeness problem* mais diretamente, conforme pode ser visto em [18, 27, 30]. Aqui os autores apresentam uma técnica chamada *event manipulation* e *artificial replay*. A idéia básica consiste em coletar informações referentes aos eventos de comunicação sincronizantes em uma execução inicial da aplicação, e assim identificar todas as condições de corrida existentes. Com isso, em uma segunda etapa, re-execuções artificiais são criadas através da manipulação das ordens dos eventos, permitindo que a aplicação tenha um outro comportamento. Os autores afirmam que conseguem investigar facilmente diferentes execuções para um mesmo conjunto de entradas. Além disso, todas as combinações possíveis de execução do programa podem ser criadas através da manipulação dos eventos, caso todas as combinações das mensagens de recebimento forem testadas. Inicialmente esse processo era manual [18], mas em [30] ele foi automatizado sem a necessidade de interação de usuários.

Algumas outras técnicas foram propostas para solucionar os problemas do não-determinismo [28, 25], mas a grande maioria está baseada em *record&replay.* Como o processo de coleta e armazenamento (*log*) das informações do programa paralelo é feito com a utilização de monitores, um monitor é basicamente um código extra (sonda) adicionado ao código fonte do programa, visando coletar informações. Essa estratégia também é conhecida como "observar um programa" [25]. O uso de monitores introduz o *probe effect,* ou *overhead* causado pela intrusão do processo de monitoração. Isso significa que a coleta de informações em tempo de execução pode influenciar o resultado do programa, seja na alteração dos tempos ou na ordem dos eventos.

Na próxima Seção um resumo das principais ferramentas existentes para a construção e análise de programas paralelos será apresentado. Inicialmente um modelo genérico para a classificação dessas ferramentas será mostrado. Logo após, estas ferramentas serão classificadas de acordo com suas principais características.

3.1 Modelo e arquitetura de ferramentas paralelas

Segundo [23], qualquer ferramenta de análise de programas paralelos consiste em dois itens principais, um componente de observação (monitoração) e um componente de análise. Outra característica importante é como e quando esses componentes interagem com a aplicação, sendo classificados em dois grupos: *on-line* e *off-line*. Quando a interação entre os componentes ocorre de maneira *on-line,* os dados do monitor são transportados durante a execução do programa para o componente de análise. Já no caso *off-line*, os dados necessários para análise são coletados pelo monitor enquanto o programa é executado, e salvos em um histórico de execução (*tracefile*) para serem analisados somente após o término da execução do programa.

Ambas as abordagens tem características diferentes. Inicialmente métodos *on-line* são mais flexíveis devido à capacidade de poder determinar e inspecionar cada estado durante a execução do programa. Por outro lado, os métodos *off-line* permitem algumas facilidades, principalmente porque ferramentas *on-line* apresentam apenas estados referentes ao passado e ao presente, enquanto ferramentas *off-line* permitem uma análise completa dos estados da aplicação feita do início até o fim da execução do programa. Um exemplo dessa diferença pode ser

percebido em técnicas para detecção de condições de corrida, onde métodos *on-line* são incapazes de identificar um conjunto completo de condições de corrida durante a execução do programa.

A maioria das ferramentas de monitoração são partes integrantes de ferramentas de análise, especialmente as ferramentas de análise *on-line*. Como exemplo é possível citar algumas ferramentas de depuração como *P2D2* [24], *PDBG* [12] e *PDT* [11] que utilizam o *GNU debugger – gdb* como depurador de baixo nível. Nesses casos, o monitor é completamente integrado no ambiente de depuração.

Quanto à capacidade de leitura dos históricos de execução (*logs* ou *tracefiles*) gerados pelas ferramentas de monitoração, a maioria dos formatos é em ASCII, o que oferece um alto grau de flexibilidade e leitura. Mas essa facilidade nem sempre é favorável, principalmente em casos em que a quantidade de dados monitorados é muito grande. Para esses casos, a solução encontrada foi criar *logs* em formatos binários, e em muitas vezes de maneira compactada, diminuindo dessa forma o seu tamanho e permitindo o armazenamento do comportamento dos programas de forma mais eficiente [47].

3.2 Resumo das principais ferramentas existentes

Baseado no modelo, nas características e nos problemas apresentados nas Seções anteriores, diversos trabalhos introduzem soluções inovadoras para tais problemas. Essas soluções correspondem ao estado da arte no contexto de ferramentas para criação, manipulação e análise de programas paralelos.

Em ambientes de teste e depuração existem trabalhos relacionados à descrição de ambientes completos que incluem a depuração como parte integral de suas estratégias de desenvolvimento. Alguns trabalhos discutem as vantagens dessa estratégia [2]. Em [35] é descrito uma combinação entre depuradores distribuídos – *DDBG* [12] com ferramentas de teste estrutural – *STEPS* [36]. Essa estratégia permite localizar erros através de análises simbólicas do código realizado com inspeções do código em execuções controladas. Uma estratégia parecida é oferecida por [54] em um ambiente integrado de ferramentas automatizadas para análise de programas que utilizam padrões de comunicação baseados em *PVM* (Tool-Set).

Outros exemplos de ambientes integrados podem ser vistos em *GRADE* (*Graphical Application Development Environment*) [26], e *MAD* (*Monitoring and Debugging Environment*) [31]. A idéia básica em *GRADE* é prover um suporte gráfico de alto nível para programação baseada em uma linguagem gráfica – *GRAPNEL* para ambientes *PVM*. Esse ambiente oferece módulos para a construção, execução, depuração, monitoração e visualização de programas paralelos baseados em troca de mensagens, podendo ser acessado pelos usuários através de uma interface gráfica. Como ferramenta de depuração, *GRADE* integra o *DDBG* como depurador distribuído.

Outro forte ambiente integrado que combina um conjunto de ferramentas para monitoração e análise de programas paralelos é o *MAD* [31]. De acordo com um modelo genérico para ferramentas de análise, os autores classificam os componentes de *MAD* em módulos de monitoração e análise. Diversas possibilidades são oferecidas em termos de monitoração. Uma implementação inicial chamada *EMU* (*Event Monitoring Utility*) gera históricos de execução para análise *off-line* em um formato de dados proprietário [32]. Uma das idéias principais é usar essa ferramenta para medir os efeitos causados na execução da aplicação com as técnicas de monitoração empregadas. Com isso, diversas estratégias foram implementadas para correção do *overhead* (intrusão) causado pelos monitores, objetivando remover a perturbação tanto no tempo de ocorrência dos eventos quanto em suas ordens.

Outra ferramenta pertencente ao *MAD*, e que se integra ao *EMU*, chama-se *PARASIT* (*Parallel Simulation Tool*), responsável por gerar execuções equivalentes em programas paralelos não-determinísticos. *NOPE* (*Nondeterministic Program Evaluator*) que representa um mecanismo completo para a técnica de *record&replay* é outra ferramenta integrada ao ambiente [30]. Ela apresenta uma característica adicional, a possibilidade de gerar manipulações automáticas de eventos com o objetivo de solucionar o *completeness problem* conforme apresentado nas Seções anteriores. Além dessas ferramentas, *MAD* apresenta ainda um módulo central – *ATEMPT* (*A Tool for Event Manipulation*), capaz de visualizar execuções paralelas utilizando diagramas baseados em técnicas de tempo e espaço (*space-time diagrams)* [33].

Para a representação gráfica do comportamento de programas paralelos, existem diversos trabalhos que descrevem abstrações de modelos comportamentais em diversos tipos de diagramas. Segundo [50], uma das melhores formas é através de Redes de Petri, o que pode ser aplicado em diferentes níveis de abstração de programas com características diferentes. Outro tipo de representação gráfica é o grafo de dependência (*dependency graph).* Ele é capaz de caracterizar visualmente dependências de dados e controle de operações executadas por um programa [5].

Existem outros tipos de representações gráficas, mas um dos mais usados é o diagrama de tempo e espaço, que foi introduzido por [34] para expressar ordem entre os eventos distribuídos. Esse diagrama é usado em muitas ferramentas para depuração paralela e ajuste de desempenho [10]. Um bom exemplo é *Paragraph*, que possui além de diagramas de tempo e espaço*,* muitas outras formas de visualização [21]. Outra ferramenta que incorporou o

diagrama de tempo e espaço foi *AIMS* [55]. *AIMS* é baseado em instrumentação em nível de código e provê portabilidade entre diferentes tipos de *hardware*. Uma ferramenta similar a *AIMS* é *XPVM* [17], e possui uma interface gráfica para análise de programas paralelos baseados em *PVM,* abstraindo graficamente tempos de computação, *overheads* de comunicação e tempos de espera, podendo ser visualizados tanto de maneira *on-line* quanto *off-line.* Uma comparação similar para programas *MPI* pode ser vista em *Upshot* [22] e *Vampir* [40], e mais recentemente em *Jumpshot4* [13]. Existem outros trabalhos com propostas semelhantes às citadas acima, entretanto uma revisão exaustiva foge ao objetivo do presente trabalho.

Essa Seção apresentou os trabalhos mais importantes e que servem de base para a definição da estratégia que está sendo proposta. Na próxima Seção, a estratégia será apresentada assim como o andamento do trabalho.

4. ESTRATÉGIA PROPOSTA

Nesse trabalho é apresentada uma estratégia cujo objetivo é possibilitar o desenvolvimento de programas paralelos mais testáveis, visando o uso de abordagens menos intrusivas nas etapas de teste. A estratégia está baseada em um processo de teste que define as etapas e técnicas utilizadas, conforme a Figura 3. Essas técnicas norteiam as contribuições mais importantes da estratégia e estão baseadas na geração de casos de teste e *scripts* para teste de *software* funcional, e na criação de um módulo de análise que teste os programas paralelos na busca por falhas na comunicação entre processos, conforme abordagens apresentadas nas Seções 2 e 3. Na Figura 4, o processo de teste é complementado com um ciclo de teste, onde suas etapas são especificadas. E na Figura 5, é definido um ambiente integrado baseado no processo de teste.

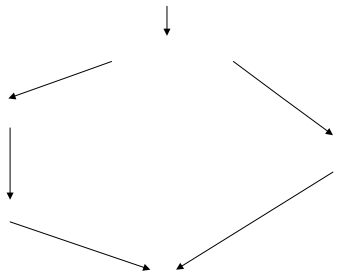


Fig. 3: Teste funcional baseado em modelos comportamentais

A geração dos casos de teste é feita automaticamente através dos modelos *SAN*, como sendo um possível ou esperado comportamento operacional da aplicação em teste. Com essa estratégia, busca-se criar através das estimativas probabilísticas dos modelos *SAN*, critérios de cobertura que possibilitem testar o sistema de maneira mais determinista. Conforme a Figura 4, o ciclo de teste é composto de diversas etapas, e algumas estão descritas a seguir:

- **Instrumentar programa:** processo de inserção de monitores no código da aplicação em teste conforme os estados determinados no modelo *SAN*. Com esses monitores no código, o módulo de análise é capaz de monitorar a execução da aplicação e identificar o conjunto de estados globais que caracterizam o comportamento observado;
- **Selecionar entradas:** etapa de leitura do *script* de teste que contem além dos casos de teste, parâmetros de execução que mapeiam os estados e autômatos do modelo com os estados e processos da aplicação;
- **Monitorar programa:** etapa de coleta do histórico de execução (*trace*) da aplicação em teste através da observação dos monitores inseridos no código. Essa etapa identifica os estados globais de execução e os eventos de comunicação;
- **Verificar a consistência de estados:** etapa que verifica se o estado global lido do modelo é atingido pela execução da aplicação.

Especifi
Progr

Modelo Compor

Programa Paralelo

Execução em um ambiente
de monitoração on-line

Comportamento Observado

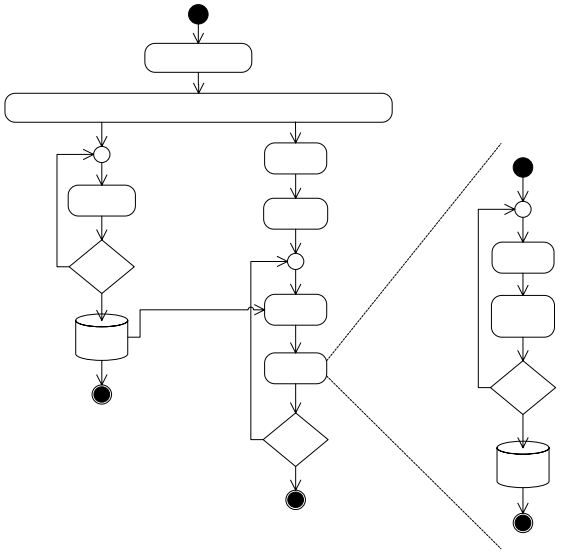


Fig. 4: Ciclo de teste do módulo de análise

Para a implementação do processo de teste, definiu-se um ambiente integrado para teste funcional de programas paralelos, conforme a Figura 5. O ambiente é composto por três módulos principais: módulo gerador de casos de teste e *scripts*; módulo de análise; e módulo de monitoração e execução, ambos desenvolvidos em JAVA. O módulo gerador de casos de teste e *scripts* estende os conceitos utilizados em [42], para gerar automaticamente casos de teste e *scripts* de teste funcional para aplicações paralelas.

O módulo de análise juntamente com o módulo de monitoração e execução, compõe o *engine* de teste e são os responsáveis pelo teste da aplicação. Conforme a Figura 5, e também às etapas especificadas no ciclo de teste, o módulo de análise recebe como estímulo (entrada), um *script* de teste contendo casos de teste e parâmetros de execução. A estrutura do *script* pode ser analisada através da Figura 7.

A interação entre os módulos e a aplicação em teste é feita de maneira *on-line*, onde os dados monitorados são transferidos para o módulo de análise em tempo de execução. Esses dados são coletados através de uma biblioteca de funções de monitoração, que é associada à execução do programa no instante da instrumentação do código da aplicação em teste. Com a integração desses módulos, é possível monitorar e analisar a execução da aplicação através dos estímulos oferecidos pelo *script* de teste, testando o comportamento da aplicação paralela na busca por situações de inconsistência entre os estados de execução e os estados do modelo. Assim, sempre que essas inconsistências forem identificadas, exceções são geradas em um histórico de resultados descrevendo as características envolvidas nos erros observados. Isso quer dizer que sempre que for identificado um estado errôneo, reporta-se o histórico de execução (*trace*) observado até esse estado atingido, tendo-se assim um conjunto de estados intermediários que ajudem na identificação da falha que causou o erro identificado.

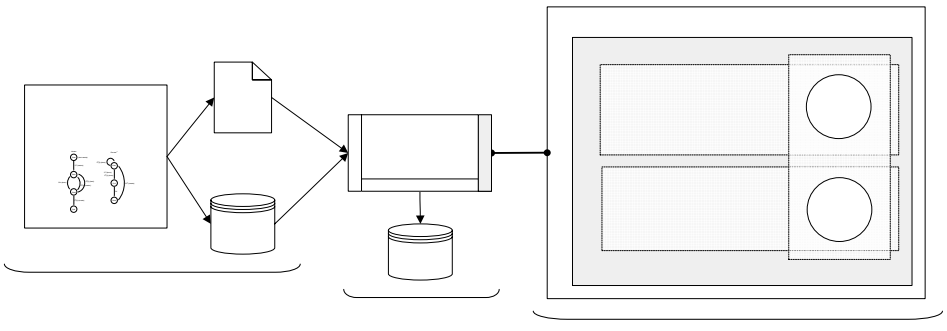


Fig. 5: Ambiente integrado para teste de programas paralelos

Como critérios inovadores, a estratégia apresenta a integração de técnicas consagradas pela comunidade científica em função de uma nova estratégia de teste, como o uso de monitores para a identificação dos estados de execução e métodos estatísticos para a geração automática de casos de teste. Um ponto forte da estratégia é a utilização de *SAN* para a representação de modelos comportamentais de programas paralelos, devido a sua

capacidade de suportar a explosão de estados ocorridos devido à complexidade desse tipo de programa. Outro ponto forte é a utilização de abordagens de depuração na investigação das falhas encontradas no teste. Com essas abordagens, é possível identificar um conjunto intermediário de estados pertencentes à execução da aplicação até o estado de erro, e dessa forma poder investigar se esse erro foi causado por falha física ou humana.

4.1 Andamento do trabalho e principais funcionalidades

A estratégia proposta apresenta-se em fase de desenvolvimento. Estudos de caso estão sendo desenvolvidos para testar a funcionalidade da estratégia, e com a avaliação desses estudos concluída, será possível identificar os **aspectos positivos e negativos** do projeto.

A **metodologia** utilizada está diretamente vinculada com as **questões de pesquisa** envolvidas na definição do processo de teste e na implementação do ambiente de teste. Busca-se com essa metodologia, um melhor embasamento da real necessidade das atividades de teste no ciclo de desenvolvimento de aplicações paralelas. Espera-se também, levantar classes de erros mais cometidos por programadores, e com isso avaliar a combinação de estratégias de teste e depuração de programas paralelos aplicadas nesse trabalho.

Inicialmente havia-se definido uma técnica de teste baseada em execuções controladas [48, 41]. Mas devido à alta intrusão causada pelas etapas de monitoração exigidas por essa técnica, uma nova abordagem foi definida, onde basicamente o engine de teste estimula a aplicação com eventos externos e espera uma lista de variáveis de estado até o estado esperado. Com isso, é possível definir quais os estados a serem testados, e principalmente estabelecer uma abordagem um pouco menos intrusiva no comportamento da aplicação em teste. A definição do conjunto de estados a serem testados deve ser feita no *script* de teste. Para isso foi estipulado o uso de um arquivo no formato xml, conforme a Figura abaixo.

```
<script_engine naut="3">

    <processors>
        <procalias alias="master" id="1"/>
        <procalias alias="slavel" id="2"/>
        <procalias alias="slave2" id="3"/>
    </processors>

    <test_case id="1">
        <global_state>
            <local_state procid="1" stateid="Start_1"/>
            <local_state procid="2" stateid="Recv"/>
            <local_state procid="3" stateid="Recv"/>
        </global_state>
        <event name="ST_1" procid="1" type="START_MASTER"/>
        <global_state>
            <local_state procid="1" stateid="Start_2"/>
            <local_state procid="2" stateid="Ord"/>
            <local_state procid="3" stateid="Recv"/>
        </global_state>
        <event name="ST_2" procid="1" type="Master"/>
        <global_state>
            <local_state procid="1" stateid="Recv"/>
            <local_state procid="2" stateid="Ord"/>
            <local_state procid="3" stateid="Ord"/>
        </global_state>
        <event name="S3" procid="1" type="QUIT_MASTER"/>
        <final_state>
            <local_state procid="1" stateid="Quit"/>
            <local_state procid="2" stateid="Ord"/>
            <local_state procid="3" stateid="Ord"/>
        </final_state>
    </test_case>


</script_engine>
```

Fig. 7: Estrutura do *Script* de Teste.

Como o engine de teste monitora constantemente a execução da aplicação para a identificação dos estados globais, o *script* de teste deve conter apenas alguns estados a serem verificados na etapa de teste, visando com isso, diminuir a intrusão do monitor na execução da aplicação.

No *script* é definido o número de autômatos, um nome para identificação dos processos, e um conjunto de casos de teste gerados. Nesse conjunto de casos de teste, devem estar definidos os estados a serem testados, os quais devem obedecer a seguinte ordem: **estado global** + **evento** + **estado global**, devendo o último estado ser definido como **estado final**. Alguns outros parâmetros estão sendo avaliados, como um critério de parada (*time-out*), um número máximo de tentativas de atingibilidade de um estado (*max_attempts*), e um controle do número máximo de mudanças de estados antes da ocorrência de um evento ou da atingibilidade de um estado respectivamente. Esses parâmetros servirão de guia para a identificação das prováveis causas dos erros identificados.

Para a avaliação dos estudos de caso, um modelo inicial de falha foi definido e está baseado em falhas de comunicação entre processos, sejam elas causadas por falhas em algum dispositivo de hardware (falhas físicas), ou

falhas humanas cometidas na codificação do programa, o que é comum de acontecer devido à complexidade dos programas paralelos.

5. CONCLUSÕES E TRABALHOS FUTUROS

Nesse trabalho foi apresentada uma estratégia cujo objetivo é possibilitar o desenvolvimento de programas paralelos mais testáveis, visando o uso de abordagens menos intrusivas nas etapas de teste. Como motivação para o desenvolvimento desse trabalho, é possível citar a necessidade existente por métodos e técnicas para testes de aplicações paralelas, algo que é percebido através das inúmeras publicações existentes, conforme as Seções 2 e 3. Outra motivação é o crescente uso de clusters computacionais, os quais necessitam de *softwar*es confiáveis.

Embora a fase de avaliação dos estudos de caso esteja no começo, problemas devido ao não-determinismo já estão sendo enfrentados. Um dos principais é o *completeness problem*, que foi discutido na Seção 3. Esse problema dificulta o estabelecimento de um critério de cobertura para a geração dos casos de teste, visto que é inviável gerar casos de teste para todos os caminhos possíveis de execução.

Como solução, optou-se por utilizar *SAN* para a representação comportamental de programas paralelos, e gerar casos de teste baseados nas probabilidades do modelo. O problema neste caso, é que devido ao não-determinismo, é muito difícil fazer com que a aplicação em teste execute esse comportamento esperado várias vezes durante a sua execução, sem que haja algum método de manipulação de eventos, conforme apresentado na Seção 3.

Diante desse problema, está sendo avaliada a implementação de um módulo para a identificação de padrões, ou perfis de comportamento. Esse padrão é conseguido através de resultados probabilísticos identificados através da observação de sucessivas execuções de um programa [41, 3]. Com isso, são criados *ranks* com as probabilidades de ocorrência dos caminhos mais e menos executados, permitindo através desses resultados, uma realimentação do modelo com esses padrões observados. Com isso espera-se testar o programa de forma mais determinista.

Como trabalho futuro, pretende-se avaliar o uso desses padrões de comportamento como critério de cobertura nos testes que estão sendo realizados nos estudos de caso em andamento.

Referências

[1] Abrams, M., Ribler, R., Mathur, A., "***Two Performance Tool Design Issues and CHITRA´S Solution",*** 1996, Symposium on Parallel and Distributed Tools, Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 98-107, ACM Press, New York, NY, USA.

[2] Appelbe, W.F., McDowell, Ch.E., ***"Integrating Tools for Debugging and Developing Multitasking Programs"***, 1989, Special issue: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Vol. **24**(1): pp. 78-88, ACM Press New York, NY, USA.

[3] Bach, J., "***A Framework for Good Enough Testing***", 1998, Computer, Vol. **31**(10): pp. 124-126, IEEE Computer Society Press, Los Alamitos, CA, USA.

[4] Baldo, L., Brenner, L., Fernandes, L.G., Fernandes, P., Sales, A., "***Performance Models for Master/Slave Parallel Programs",* 2004,** First Internationl Workshop on Practical Applications of Stochastic Modeling.

[5] Beguelin, A., Dongarra, J.J., Geist, A., Sunderam, V.S., ***"Visualization and Debugging in a Heterogeneous Environment"***. 1993, IEEE Computer, Vol. **26**(6): pp. 88-95, IEEE Computer Society Press, Los Alamitos, CA, USA.

[6] Bertolini, C., Farina, A. G., Fernandes, P., Oliveira, F. M., "***Test Case Generation Using Stochastic Automata Networks: Quantitative Analysis",*** 2004, Second International Conference on Software Engineering and Formal Methods (SEFM´04), pp. 251-260, IEEE Computer Society, Beijing, China.

[7] Bezerra, E. A., Vargas, F., Gough, M. P., ***"Improving reconfigurable systems reliability by combining periodical test and redundancy techniques: a case study"***, 2001, journal of Electronic Testing: theory and Applications, Vol.**17**(3): pp 701-711, Jetta, Norwell, Ma, Usa.

[8] Bezerrra, E. A., Jansch-Porto, I., "***Test procedure for faults detection in the transputer processor"***, 1997, Journal of solid State Devices and Circuits, Vol. **5**(1): pp 27-27, São Paulo.

[9] Boehm, B.W., ***"Software Engineering"****, 1976,* IEEE Transactions on Computer, Vol. **25**(12): pp. 1226-1241.

[10] Browne, J.C., Hyder, S.I., Dongarra, J., Moore, K., Newton, P., "***Visual Programming and Debugging for Parallel Computing"****,* 1995, IEEE Parallel & Distributed Technology: System & Technology, Vol. **3**(1): pp. 75-83, IEEE Computer Society Press, Los Alamitos, CA, USA.

[11] Clemencon, C., Fritscher, J., Rühl, R., "***Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool"***. 1995, High Performance Computing Symposium, Montreal, Canada.

[12] Cunha, J.C., Lourenço, J.M., Vieira, J., Moscão, B., Pereira, D., ***"A Framework to Support Parallel and Distributed Debugging"***, 1998, Proceedings of the International Conference and Exhibition on High-Performance Computing and

Networking, Lecture Notes In Computer Science, Vol. 1401: pp. 708-717, Springer-Verlag, London, UK.

[13] Chan, A., Gropp, W., Lusk, E., "***Scalable Log Files for Parallel Program Trace Data*** *(DRAFT)"*, in *IL 60439*. 2000, Argonne National Laboratory: Argonne.

[14] Farina, A.G., Fernandes, P., Oliveira, F.M, ***"Representing Software Usage Models with Stochastic Automata Networks"***, 2002, Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, Vol. 27, ACM Press, New York, NY, USA.

[15] Foster, I.T., "***Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering****",* 1995, p. 430, Addison-Wesley, Boston, MA, USA.

[16] Fujiwara, S., Bochmann, G. V., Khendek, F., Amalou, M., Ghedamsi, A., ***"Test Selection Based on Finite State Models"****,* 1991, IEEE Transactions on Software Engineering, Vol. **17**(6): pp. 591-603, IEEE Press, Piscataway, NJ, USA.

[17] Geist, G.A., Kohl, J., Papadopoulos, P., "***Visualization, Debugging, and Performance in PVM"***, 1996, Proceedings. of Visualization and Debugging Workshop, in Debugging and Performance Tuning for Parallel Computing Systems, IEEE Computer Society Press, Los Alamitos, CA, USA.

[18] Grabner, S., Volkert, J., "***Debugging Distributed Memory Programs Using Communication Graph Manipulation****", 1996, International Symposium on High Performance Computing Systems*, Canada.

[19] Hartmann, J., Robson, D. J., "***Techniques for Selective Revalidation",*** 1990, IEEE Software, Vol. **7**(1): pp. 31-36, IEEE Computer Society Press, Los Alamitos, CA, USA.

[20] Hayes, A.H., Simmons, M.L., Brown, J.S., ***"Debugging and Performance Tuning for Parallel Computing Systems"****,* 1996, p. 400, IEEE Computer Society Press, Los Alamitos, CA, USA.

[21] Heath, M.T., *"****Recent Developments and Case Studies in Performance Visualization using ParaGraph"***, 1993, Workshop on Performance Measurement and Visualization of Parallel Systems, pp. 175-200, Elsevier Science Publishers, B. V. Amsterdam, The Netherlands, The Netherlands.

[22] Herrarte, V., Lusk, E., "***Studying parallel program behavior with Upshot"***, in *Technical Report ANL-91/15*. 1991, Argonne National Laboratory, IL, USA.

[23] Hondroudakis, A., "***Performance Analysis Tools for Parallel Programs",*** 1995, E.P. Computing Centre, The University of Edinburgh, (http://www.epcc.ed.ac.uk/epcc-tec/documents.html).

[24] Hood, R, "***The p2d2 Project: Building a Portable Distributed Debugger",*** 1996, ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 127-136, ACM Press, New York, NY, USA.

[25] Kacsuk, P, ***"Systematic Debugging of Parallel Programs Based on Collective Breakpoints"***, 1999, International Symposium on Software Engineering for Parallel and Distributed Systems. P. 83, IEEE Computer Society, Washington, DC, USA.

[26] Kacsuk, P., Cunha, J.C., Dozsa, G., Lourenco, J., Fadgyas, T., Antao, T., "***A Graphical Development and Debugging Environment for Parallel Programs", 1997,*** Journal of Parallel Computing, Distributed and Parallel Systems: Environments and Tools, Vol. **22**(13): pp. 1747-1770, Elsevier Science Press.

[27] Kranzlmüller, D., Grabner, S., and Volkert, J., *"****Using Control and Data Flow Analysis for Race Evaluation"***, 1997, Proceedings of the Third International Euro-Par Conference on Parallel Processing, Eds. Lecture Notes In Computer Science, Vol. 1300, pp. 102-109, Springer-Verlag, London, UK.

[28] Krawczyk, H., Krysztop, B., Proficz, J, ***"Suitability of the Time Controlled Environment for Race Detection in Distributed Applications"***, 2000, Future Generation Computer Systems, Special Issue on Distributed and Parallel Systems, Vol. **16**(6): pp. 625-635, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands.

[29] Krawczyk, H., Wiszniewski, B, "***Interactive Testing Tool for Parallel Programs****",* 1996, Software Engineering for Parallel and Distributed Systems, Chapman Hall, London, UK.

[30] Kranzlmüller, D., Volkert, J., "***Debugging Point-to-Point Communication in MPI and PVM***", 1998, Proceedings of the 5th European PVM/MPI Users Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes In Computer Science, Vol.1497: pp. 265-272, Springer-Verlag, London.

[31] Kranzlmuller, D., Grabner, S., Volkert, J., "***Debugging with the MAD environment"***, 1997, Parallel Computing, Special double issue on environment and tools for parallel scientific computing, Vol. 23 (1-2): pp. 199-217, Elsevier Science Publishers, Amsterdam, The Netherlands.

[32] Kranzlmüller, D., Grabner, S., J. Volkert, J., "***Monitoring Strategies for Hypercube Systems***.", 1996, Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP ´96), pp. 486, IEEE Computer Society, Washington, DC, USA.

[33] Kranzlmüller, D., Grabner, S., Volkert, J., ***"Message Passing Visualization with ATEMPT"***. 1995, Parallel Computing: State-ofthe-Art and Perspectives*, in: Proc. ParCo95, Conference on Parallel Computing*, Gent, Belgium.

[34] Lamport, L., "***Time, Clocks, and the Ordering of Events in a Distributed System****",* 1978, Communications of the ACM, Vol. **21**(7): pp. 558-565, ACM Press, New York, NY, USA.

[35] Lourenco, J., Cunha, J.C. Krawczyk, H., Kuzora, P., Neyman, M., Wiszniewski, B., ***"An Integrated Testing and Debugging Environment for Parallel and Distributed Programs"***, 1997, Proceedings of the 23rd EUROMICRO Conference (EUROMICRO'97), pp. 291-298, IEEE Computer Society Press, Budapest, Hungary.

[36] Leu, E., Schiper, A., Zramdini, A. *"**Execution Replay on Distributed Memory Architectures**"*, 1990, Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, pp. 106-112.

[37] Murphy, G.C., Townsend, P. S., ***"Experiences With Cluster and Class Testing",*** 1994, Communications of the ACM, Vol. **37**(9): pp. 39-47, ACM Press, New York, NY, USA.

[38] Netzer, R.H.B., Miller, B.P, ***"Optimal Tracing and Replay for Debugging Message-Passing Parallel Program"***, 1992, Conference on High Performance Networking and Computing, Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, pp. 502-511, IEEE Computer Society Press Los Alamitos, CA, USA.

[39] Netzer, R.H.B., Brennan, T.W., Damodaran-Kamal, S.K., ***"Debugging Race Conditions in Message-Passing Programs"***, 1996, Symposium on Parallel and Distributed Tools, Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 31-40, ACM Press, New York, NY, USA.

[40] Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.-C., Solchenbach, K. ***"VAMPIR: Visualization and Analysis of MPI Resources"***. in *Supercomputer*. 1996.

[41] Oberhuber, M., ***"Elimination of Nondeterminacy for Testing and Debugging Parallel Programs"***, 1995, International Workshop on Automated and Algorithmic Debugging, Saint Malo, France.

[42] Oliveira, F.M., Copstein, B., Reginato, L. R.C, *"**STAGE: an Integrated Environment for Statistical Test Script Generation**"*, 2004, Workshop de Testes e Tolerância a Falhas. . Gramado, RS, Brasil.

[43] Paul, J., *"**Software Engineering - General Testing and Debugging Guidelines**"*, 1999, ttp://www.jodypaul.com/SWE/TD/TestDebug.html).

[44] Pradhan, D. K., *"**Fault-Tolerant Computer System Design**"*, 1994, p. 550, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[45] Rosenberg, J.B., ***"How Debuggers Work: Algorithms, Data Structures, and Architecture",*** 1996, p. 256, John Wiley & Sons, Inc., New York, NY, USA.

[46] Reed, D.A., Nickolayev, O.Y., Roth, P.C., ***"Real-Time Statistical Clustering for Event Trace Reduction"***, 1997, The International Journal of Supercomputer Applications and High Performance Computing, Vol. **11**(2): pp. 144-159.

[47] Ronsse, M.A., Levrouw, L.J., Bastiaens, K., *"**Efficient Coding of Execution-Traces of Parallel Programs**"*, 1995, *Proceedings of the ProRISC / IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, Utrecht.

[48] Tai, K.C., Carver, R.H., ***"Testing Distributed Programs"*** *(cap. 33).* in: Zomaya, A.Y., (Ed.), "Parallel and Distributed Computing Handbook", 1996, p. 1232, Mcgraw-Hill Computer Engineering Series, New York, NY.

[49] Valadan, G.S., ***"Trend´s in Reliability and Test Strategies,*** *1995,* IEEE Software, Vol. **12**(3).

[50] Vautherin, J., ***"Parallel Systems Specifications with Coloured Petri Nets and Algebraic Abstraction Data Types",*** 1987, Advances in Petri Nets, 7th European Workshop on Applications and Theory of Petri Nets, pp. 293-308, Springer-Verlag, Berlim.

[51] Wasserman, H., Blum, M., *"**Software Reliability via Run-Time Result-Checking",*** 1997, Journal of the ACM (JACM), Vol. **44**(6): pp. 826-849, ACM Press, New York, NY, USA.

[52] Walton, G.H., Poore, J.H., Trammell, C.J., *"**Statistical Testing of Software Based on a Usage Model",*** 1995, Software-Practice & Experience, Vol. **25**(1): pp. 97-108, John Wiley & Sons, Inc., New York, NY, USA.

[53] Whittaker, J.A., Thomason, M.G., *"**A Markov Chain Model for Statistical Software Testing***", 1994, IEEE Transactions on Software Engineering on Special Section on the 15th Annual Internacional Conference on Software Engineering, Vol. **20**(10): pp. 812-824, IEEE Press , Piscataway, NJ, USA.

[54] Wismüller, R., Ludwig, T., Bode, A., Borgeest, R., Lamberts, S., Oberhuber, M., Röder, C., Stellner, G., *"**The TOOL-SET Project: Towards an Integrated Tool Environment for Parallel Programs***", 1997, Proceedings of Second Sino-German Workshop on Advanced Parallel Processing Technologies - APPT'97, pp. 9-16, Verlag Dietmar Folbach, Koblenz, Germany.

[55] Yan, J.C., Jin, H.H., Schmidt, M.A., *"**Performance Data Gathering and Representation from Fixed-Size Statistical Data",*** 1998, *Technical Report NAS-98-003, in: http://www.nas.nasa.gov/Research/Reports/Techreports/1998/nas-98-003.pdf*, NAS Systems Division, NASA Ames Research Center, Moffet Field, California, USA.

Anexo B

ARTIGO - LATW 2006

Trabalho aprovado a ser apresentado no "*7th IEEE Latin-American TestWorkshop* (LATW 2006)", a ser realizado de 26 a 29 de março de 2006, em Buenos Aires, Argentina.

Probe Effect Mitigation in the Software Testing of Parallel Systems*

Leonardo Albernaz Amaral [1], Eduardo Augusto Bezerra [2], Flavio Oliveira [1],
Luiz Gustavo Fernandes [3], Mateus Raeder [3] and Pedro Velho [3]

Software Testing Research Centre (CPTS), Hewlett-Packard /PUCRS Cooperation [1]
Embedded Systems Research Centre (CPSE), Hewlett-Packard/PUCRS Cooperation [2]
Parallel Applications Research Centre (CAP), Hewlett-Packard /PUCRS Cooperation [3]
Informatics Faculty, PPGCC, Catholic University of RS (PUCRS), Porto Alegre, RS, Brazil
{lamaral, bezerra, flavio, gustavo, mraeder, pedro}@inf.pucrs.br

Abstract

This work introduces a less intrusive software testing strategy (probe effect mitigation) for parallel programs. In the proposed strategy, a discrete stochastic formalism (SAN) is used for the behavioural modelling of parallel applications. Test cases are created from this behavioural model to stimulate parallel programs, searching for inter-process communication errors. An important contribution is the generation and use of SAN-based test cases to reach coverage criteria, according to probabilities obtained from behavioural rates defined in the application model. In order to define a less intrusive test strategy, emphasis has been given not only to the definition and generation of test cases, but also in the approaches employed in the test engine for the monitoring stages.

1. Introduction

A parallel program can be seen as a collection of communicating sequential processes that run simultaneously [1]. From the test and debug point of view, they show several difficulties when compared to the sequential ones. These difficulties arise mainly from inter-process communication and synchronisation activities, and one of the hardest challenges is the non-determinism problem [2]. A non-deterministic behaviour is distinguished by the breach of the execution determinism, i.e., diverse runs of the same program may spawn distinct synchronizing sequences, resulting in several execution paths [3]. There are two problems related to the non-determinism: the irreproducibility effect and the completeness problem. The first one is characterised as the inability of performing cyclic debugging, i.e., the impossibility of reproducing an error in the test phase, as the program may follow different paths in each subsequent execution. The second one, completeness problem, concerns test coverage criteria, as it is perceived when testing a parallel program's behaviour in all of its possible execution paths. Considering the amount of paths to be verified the test coverage could become impracticable.

Statistical testing has been calling the attention in recent years, as it overcomes weaknesses found in other strategies. Usually, it is based on use models that represent possible actions that a user can take when modelling the behaviour of a specific piece of software. Usually, the used formalisms are based on state graphs representing transition probabilities. The first formalisms used in the statistical testing were the Markov Chains. However, recent studies have been conducted around another sort of formalism named Stochastic Automata Networks (SAN). As models grow in size and complexity of their components, Markov Chains show limitations in their capacity, and the SAN model offers better resources for the program representation.

This work introduces a strategy of software testing for parallel programs, aiming the mitigation of the intrusion caused by the software testing activities. This strategy is based on the idea that it is possible to employ discrete formalism (SAN) for the behavioural model representation of parallel applications, allowing the mapping of the model states to the application code (code instrumentation). This will define the mapping of SAN model events to the application events. As part of the proposed strategy, test cases and scripts for the functional testing of the application can be generated by the statistical refining of the usage estimation defined in the model. **An important feature of the strategy is the statistical definition of the most used parts of parallel programs, reducing the complexity of their testing by employing not only traditional software testing approaches, but also parallel program debugging techniques.**

This paper is organized as follows. Section 2 describes the formalism for the proposed strategy. Section 3 introduces the testing and debugging of parallel programs. Section 4 describes the proposed strategy and a case study. Section 5 discusses some conclusions and future work.

2. Formalism for Statistical Testing

SAN is a formalism capable of modelling a system consisting of several sub-systems, i.e., a

* This work was developed in collaboration with HP Brazil R&D.

system made of quasi-independent modules. Each of the sub-systems is represented by a stochastic automaton and by the transition from one state to another in this automaton. An automaton is a set of states and its rules, or probability functions, defining changes from a state to another. The "local state" of an automaton at time *t* is its current state. The SAN "global state" is given by the "local state" that each of its automaton occupies at time *t*. SAN events can be local or synchronizing, having the capability of changing the global state of a SAN. The local events change the state of only one automaton. Synchronizing events make changes to more than one local state, i.e., they promote state changes in more than one automaton at the same time.

Fig. 1 shows the use of SANs in the modelling of users behaviour during system authentication. Its structure has the following components:

- Automata (2): {*Login Automaton*, *Password Automaton*};
- States (6): {*Start*, *Menu*, *Password*} and {*PNotOk*, *Waiting*, *POK*};
- Events (5): {*ST*, *QT*, *S*} synchronizing events and {*g*, *f*} local events;

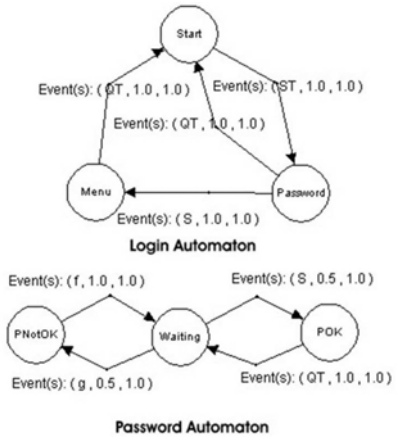


Fig. 1. SAN model of a login system.

The system behaviour can be observed in the two test cases for this model shown in Fig. 2. The network global states (e.g. lines 2, 4, 11, 13) and the event that generates the state transition (e.g. lines 3, 12) are presented step-by-step in the test cases. Every time a synchronizing event takes place (e.g. lines 3, 10) the automata states of the event are modified simultaneously. However, when local events take place (e.g. lines 12, 14) only local states pertaining to the event are changed. The original definition of SANs is continuous-time. In order to allow a discrete interpretation, necessary for test case generation, restrictions have been introduced to the model, such as initial and final states/events and global event probabilities derived from normalization of event rates.

01: **[TestCase1]**
02: *LoginAutomata.Start, PasswordAutomata.Waiting*
03: *START_MASTER EVENT ST Executed*
04: *LoginAutomata.Password, PasswordAutomata.Waiting*
05: *QUIT_MASTER EVENT QT Executed*
06: *LoginAutomata.Start, PasswordAutomata.Waiting*
07:
08: **[TestCase2]**
09: *LoginAutomata.Start, PasswordAutomata.Waiting*
10: *START_MASTER EVENT ST Executed*
11: *LoginAutomata.Password, PasswordAutomata.Waiting*
12: *LOCAL EVENT g Executed*
13: *LoginAutomata.Password, PasswordAutomata.PNotOK*
14: *LOCAL EVENT f Executed*
15: *LoginAutomata.Password, PasswordAutomata.Waiting*
16: *QUIT_MASTER EVENT QT Executed*
17: *LoginAutomata.Start, PasswordAutomata.Waiting*

Fig. 2. Test Cases from the SAN model in Fig. 1

The test cases in Fig. 2 were produced by STAGE, which is an integrated environment for test cases creation and for the generation of statistical testing scripts [4, 5]. STAGE has been developed at CPTS/PUCRS, in cooperation with HP Brazil. Its main features are the use of SAN for use model representation, and the proposal of an intermediary model for mapping the system component interfaces to the use model. SAN allows complex and non-deterministic systems to be represented in a modular way, reducing the explosion of state space typical of Markov Chains.

In this work the idea is to extend STAGE in order to deal with the complexity of parallel programs, contributing to the development of an integrated environment for the modelling and automatic generation of test cases and scripts for the functional software testing. In a related work, Beguelin et al. [6] employ statistical test techniques based on Markov Chains for the formal representation of the application's use model. In [7], several SAN models were generated and compared to Markov Chain models considering the number of states, scalability and facility for a user to read the models. In this comparison, SAN show to be superior. In [8], test cases were automatically generated from SAN models.

3. Testing Parallel Programs

According to [9], any parallel programming analysis tool is made of two main items, an observation component (monitoring), and an analysis component. Another important feature is how and when these components interact with the application, which could be *on-line* or *off-line*. For the former, monitor data are transferred to the analysis component, during the program

execution. For the off-line, data are collected by the monitor while the program runs, and stored in an execution log (*tracefile*). Afterwards, when the program's execution is complete, the stored data can be analysed.

In general, on-line methods are more flexible as they make easy the inspection of each state during program execution. On the other hand, off-line methods introduce some facilities, mainly because on-line tools deal only with present and past states, while off-line tools allow a more complete analysis of the application states from the beginning till the end of the program. A good representative of this situation can be perceived in race condition detection techniques, where on-line methods are not capable of identifying a full set of race conditions during the program's execution.

Several techniques have been proposed targeting the non-determinism/irreproducibility effect problem [9], and most of them are based on *record&replay*. This mechanism has two stages. The first one is the recording phase, where the messages order is stored in an execution log (*tracefile*), which keeps records of all non-deterministic events. In the second stage takes place the replay phase, where the stored data are used to produce re-executions according to the previously observed ones.

The process of collecting and logging information concerning the parallel program is done by monitors, which are extra pieces of code added to the program's source code aiming the collection of information. This strategy is also known as program observation [9]. Monitors introduce the ***probe effect***, or *overhead* resulting from the intrusion of the monitoring process. This mean that the runtime information collection may affect the program's result, either in the timing or in the event order. This problem is harder in statistical testing, due to the large sample sizes, which imply automated execution.

Some techniques have been proposed also to deal with the completeness problem mentioned before, *"controlled execution"* and *"event manipulation & artificial replay"*. Controlled execution is a method for automatic testing conduction, where control patterns are used to establish orders among communication events [10].

4. Proposed Strategy

In state-based functional software testing, a program can be seen as a set of states and their transitions, defining the system behaviour during its execution. The test strategy validation is performed by the consistency verification of the observed behaviour during the program's execution, against its expected behaviour, usually based on the program specification.

As the present work is based on the use of discrete formalism models for the behavioural representation of parallel applications, it justifies the use of SANs for this abstraction.

4.1. Testing Methodology

Fig. 3 shows the whole testing methodology, where the program testing activity is explored in more details.

The test case generation is done automatically through the SAN models, as a possible or expected operational behaviour of the application under test. Applying this strategy, the aim is to create coverage criteria, through probabilistic estimations of SAN models, which allow the system testing in a more deterministic way. As shown in Fig. 3, the testing cycle has several steps.

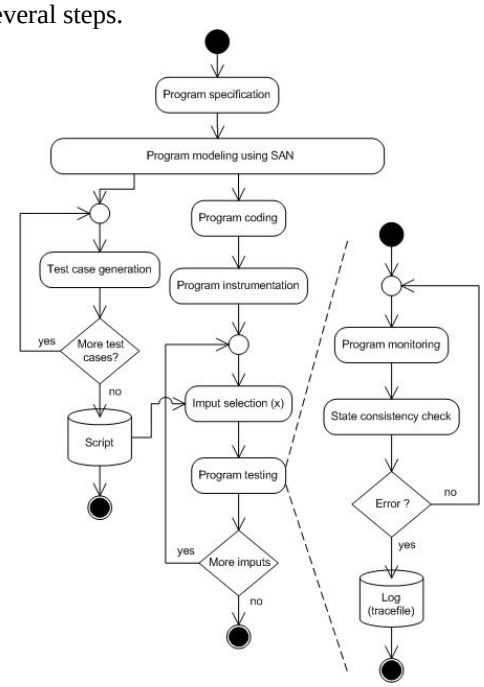


Fig. 3. Testing methodology.

Some of these steps are the following: **program instrumentation** - process of inserting monitors in the code of the application under test, according to the states defined in the SAN model. Having these monitors in the code, the analysis module is capable of monitoring the application's execution, and identify the set of global states that better represent the observed behaviour; **input selection** - in this stage the test script is read. This script lists not only the test cases, but also the execution parameters that map the states and model automata to the application states and process; **program monitoring -** trace execution log for the application under test through the

observation of the monitors inserted in the code. This step identifies the global states of execution and the communication events; and **state consistency check -** this stage checks whether the global state obtained from the model is reached by the application.

The environment has three main modules: the **test cases and scripts generation module**; the **analysis module**; and the **execution and monitoring module**, both written in Java. The test cases and scripts generation module extends the concepts used in [5] for the automatic generation of test cases and scripts for the functional testing of parallel applications.

An integrated environment for the software testing of parallel programs has been defined, as shown in Fig. 4.

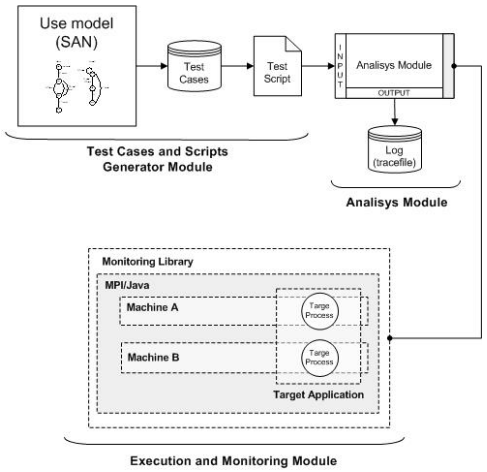


Fig. 4. Proposed integrated environment.

The analysis module, together with the monitoring and execution modules, represent the test engine and are the responsible for the test application. According to Fig. 4, and also to the steps specified in the test cycle, the analysis module receives as stimulus (input), a test script with the testing cases and execution parameters.

The interaction between the modules and the application under test is done in an on-line way, where the monitored data are transferred to the analysis module at run time. These data are collected using a library of monitoring functions, which is associated to the program execution during the code instrumentation of the application under testing. This is the test procedure to check the parallel application behaviour searching for situations of inconsistency between the execution states and the model states. Thus, every time an erroneous state is detected, there is a record of the execution trace observed until this state is reached. The result is a set of intermediary states that help in the identification of the fault that caused the detected error.

The innovation introduced by the strategy includes the integration of traditional techniques in a new test strategy, the use of monitors in the identification of execution states, and statistical methods for the automatic generation of test cases. An important feature of the strategy is the use of SAN for the representation of behavioural models. It is also important to mention the use of debugging approaches in the investigation of the faults detected by the test. By employing these approaches it is possible to identify an intermediary set of states belonging to the application execution, throughout the erroneous state and, in this way, allowing the investigation whether the error has human or physical origin.

4.2. Main Functionalities

The proposed strategy is an on-going work. Case studies have been developed in order to validate the strategy's functionality, allowing the identification of the positive and negative aspects of the project.

Initially, it has been defined a test technique based on controlled executions [11]. As a result of the high degree of intrusion caused by the monitoring stages, according to the technique demands, a new approach was adopted where, basically, the test engine exercises the applications through external events and waits for a list of state variables, until the expected state is reached. This allows the definition of which states will be tested and, mainly, in the establishment of an approach that would be less intrusive in the functionality of the application under test. The definition of the set of states to be tested should be defined in the test script, and an XML format has been defined for this file. As the test engine keeps monitoring the application execution in order to identify the global states, the test script should have only some states to be verified in the test stage. The objective is the reduction of the intrusion introduced by the monitor during the application execution.

The script defines the amount of automata, a label for the processes identification, and a set of generated test cases. States to be tested should be defined in the set of test cases, respecting the following order: global state + event + global state. Further parameters are under evaluation as, for instance, a stop criterion (*time-out*), a maximum number of attempts to reach a state (*max_attempts*), and a control of the maximum number of state transitions before the manifestation of a state or before a state is reached, respectively. These parameters will be used to help in the identification of causes for the observed errors.

4.3. Case Study Evaluation

In parallel and distributed systems, usually, it is assumed a fault model based on the following: collapse, fail stop, send/receive omission, network failure, network partitioning, timing fault, and Byzantine faults. As this work's goal is to test the behaviour of parallel applications based on message exchange, and also considering an easily controlled execution environment (e.g. a Cluster application based on Message Passing Interface – MPI), the selected **fault model** takes into consideration the following:

- states defined in the model cannot be reached;
- States not allowed in the model are reached;
- Deadlocks and livelocks (time out);
- Performance faults (communication bottlenecks); and
- Omission faults.

Initially, faults regarding state and events accomplishment are automatically identified. Other sort of faults should be identified manually through the tracefile, which has the set of states and events reached during the program's execution until the moment the fault took place.

The test engine functionalities have been validated by employing a set of test cases for a couple of non-deterministic applications. This work describes the results obtained from the *multiple-vector sorting* application, using the Bubble sort algorithm. It is important to recall that the functionalities to be validated in this case study are: the test engine capability of reaching all states during the execution of the application under test; and a performance comparison between the engine and another monitoring tool, aiming the validation of the intrusion mitigation. Other functionalities as fault coverage and statistical validation are not considered in this case study.

First results, considering the selected test data, show that the engine has reached 100% of the expected states, i.e., all the expected states have been reached by the application during its execution by the engine.

For the proposed approach performance evaluation, the main issue concerns the **probe-effect** introduced in the application's behaviour. A special process called "engine" is added to the parallel application acting as a monitoring module. In addition, each application process must be modified to explicitly send information about state transitions. These modifications should be made by the use (developer). The program is then executed recording the result in a log file which can be analyzed off-line, in case the on-line test point out an error.

Traditionally, the MPI library already provides off-line monitoring with the Multi-Processing Environment (MPE). MPE provides a way to log state transitions and synchronization of processes which can be visualized after the execution. Our approach presents the advantage of automatically detection (on-line testing) when an inconsistent state is reached with no need for a post-analysis.

In this context, it is natural to choose MPE as the comparison parameter in order to evaluate the probe-effect of the proposed approach. In the chosen multiple-vector sort problem, **twenty vectors** are sorted in parallel in a manager/worker approach. The manager process is responsible for distributing the vector on demand, *i.e.*, each time a process becomes idle it receives a new vector to be sorted. The remaining processes (the workers) behave as follows: they receive an unsorted vector, they sort it and they send it back to the manager. The workers keep doing these activities in a loop until they receive a control message indicating that there is no more work left.

Tests were carried out using a cluster of 14 workstations each one featuring a Pentium III 500 MHz processor, 256 MB RAM interconnected through a 100 Mb FastEthernet link. Three different vector sizes have been used: 5,000, 10,000, and 20,000 integer elements. The number of worker processes ranges from 3, 5, 7, 9 and 11. The curves plotted in Figs. 5, 6, and 7 are the result of the average of five executions for each point. For the sake of clarity, each test vector size is represented in a different graph where the x axis is the number of workers and the y axis is the resulting execution time in seconds. Each graph has four curves, one for each kind of implementation/execution: pure C; pure Java; C plus MPI/MPE; and Java plus MPI and the proposed engine.

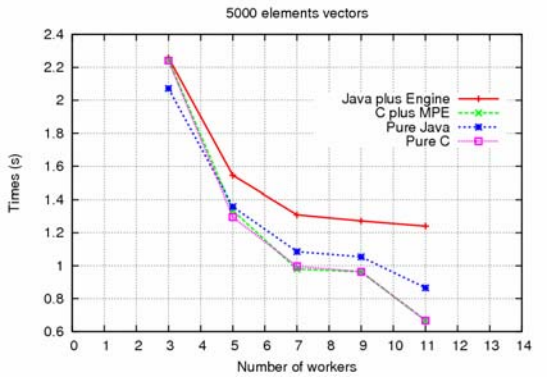


Fig. 5. Probe-effect for 5,000 elements vectors.

Fig. 5 presents the results for 5,000 elements vectors. In this case, the C/MPE version presents a better performance. Also, it is possible to observe that the probe-effect in the Java MPI code becomes more important as the number of workers grows.

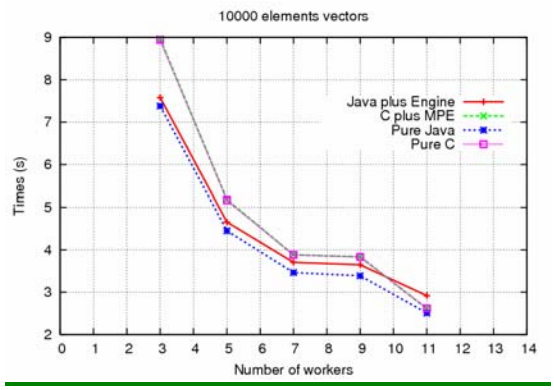


Fig. 6. Probe-effect for 10,000 elements vectors.

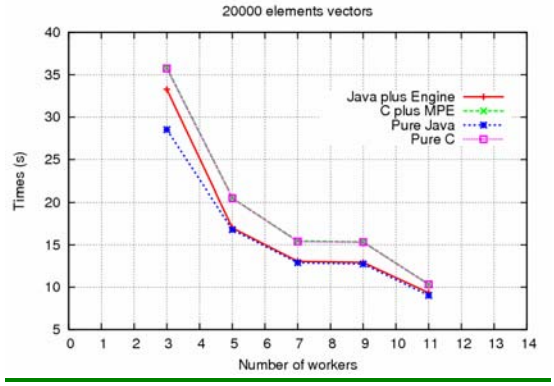


Fig. 7. Probe-effect for 20,000 elements vectors.

According to Figs. 6 and 7, the Java MPI plus engine version becomes faster than the C version in larger problems. A possible reason is that when data increase, the amount of computation becomes more significant than the probe-effect. The exception for that is the eleven workers configuration for the 10,000 elements vectors in Fig. 6. In that case, the amount of computation of one worker process is not enough to make up for the overhead introduced by the JVM plus the engine.

Finally, it is important to highlight that, as the problem size grows, the probe-effect in our approach becomes almost non-significant (see Fig. 7).

5. Conclusion

The proposed strategy aims the improvement of parallel programs testability, during the development phase, by using less intrusive approaches in the test stages.

The intrusion diminution is applied to several stages in the test process, but the case study considers only the approach regarding the proposed monitoring method, which is usually less intrusive than MPE itself. Other approaches are related to the decrease in the amount of states and events to be used in test cases, and the decrease in the number of monitors in the application code. Both approaches do not have an impact on the test strategy.

A couple of problems related to the non-determinism, to be more precise, to the completeness problem, have been identified. For instance, the detection of "not a bug" errors, which were not suppose to be detected. A way of going around this problem, is the use of execution patterns to feedback the SAN model in the scripts generation module (see Fig. 4). As a result, the strategy can generate test cases (paths) having a higher probability of happening, reducing then false error detections.

SAN model feedback based on execution patterns and the creation of grouped test cases, are new approaches to be followed in the research of solutions for the non-determinism in the testing of parallel applications.

References

[1] Hoare, C. A. R. ***Communicating Sequential Processes***. Comm. of the ACM, v.21, n.8, p.666-677, NY, 1978.

[2] Appelbe, W.F., McDowell, C.E., ***Integrating Tools for Debugging and Developing Multitasking Programs***, Proc. of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, v. **24**(1): pp. 78-88, NY, 1999.

[3] Hayes, A. H. et al., ***Debugging and Performance Tuning for Parallel Computing Systems***, 400 p., IEEE Computer Society, 1996.

[4] Farina, A.G., et al. ***Representing Software Usage Models with Stochastic Automata Networks***, Proc. 14th Int. Conf. on Software Engineering and Knowledge Engineering, Italy, v. 27, ACM Press, NY, 2002.

[5] Oliveira, F.M. et al. ***STAGE: an Integrated Environment for Statistical Test Script Generation*** Test and Fault Tolerance Workshop, Gramado, Brazil, 2004.

[6] Beguelin, A. et al. ***Visualization and Debugging in a Heterogeneous Environment*** Computer, v. **26**(6): pp. 88-95, IEEE, Los Alamitos, 1993.

[7] Fujiwara, S. et al. ***Test Selection Based on Finite State Models*** IEEE Trans. on Software Engineering, v. **17**(6): pp. 591-603, NJ, 1991.

[8] Bertolini, C. et al. ***Test Case Generation Using Stochastic Automata Networks: Quantitative Analysis*** 2nd Int. Conf. on Software Engineering and Formal Methods, pp. 251-260, IEEE, Beijing, China, 2004.

[9] Krawczyk, H., et al. ***Suitability of the Time Controlled Environment for Race Detection in Distributed Applications*** Future Generation Computer Systems, Special Issue on Distributed and Parallel Systems, v.**16**(6): pp. 625-635, Elsevier, Amsterdam, 2000.

[10] Hood, R. ***The p2d2 Project: Building a Portable Distributed Debugger*** ACM Sigmetrics Symp. on Parallel and Distributed Tools, pp. 127-136, ACM Press, NY, 1996.

[11] Oberhuber, M. ***Elimination of Nondeterminacy for Testing and Debugging Parallel Programs*** Int. Workshop on Automated and Algorithmic Debugging, Saint Malo, France, 1995.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo