Contribuições na área de Teste de Software Concorrente

Simone do Rocio Senger de Souza

Contribuições na área de Teste de Software Concorrente

Simone do Rocio Senger de Souza

Texto sistematizando o trabalho científico da candidata, apresentado ao Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, como parte dos requisitos para obtenção do Título de Professor Livre Docente, junto ao Departamento de Sistemas de Computação.

USP - São Carlos
Janeiro/2014

Sumário

and reachability testing to improve concurrent program testing quality. In: 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE2011), Miami Beach, USA, 2011d, p. 207-212. 127

F (Silva et al. 2012b) Silva, R. A.; Souza, S. R. S.; Souza, P. S. L. Mutation Operators for Concurrent Programs in MPI. In: 13th Latin American Test Workshop, Quito, Ecuador, 2012b, p. 69-74 137

G (Sarmanho et al. 2008) Sarmanho, F. S.; Souza, P. S. L.; Souza, S. R. S.; Simao, A. S. Structural testing for semaphore-based multithread programs. In: ICCS - International Conference on Computational Science, Lecture Notes in Computer Science, 5101, Krakow: Springer-Verlag, 2008, p. 337-346. 145

Capítulo

1

Introdução

Este capítulo descreve o contexto das linhas de pesquisa nas quais se inserem os trabalhos desenvolvidos pela autora, apresentando os desafios principais que motivaram as pesquisas realizadas e em desenvolvimento.

1.1 Contextualização

A Engenharia de Software é uma disciplina que aplica os princípios de engenharia com o objetivo de produzir software de alta qualidade a baixo custo e, para isso, diversos processos, métodos, técnicas e ferramentas foram propostos (Pressman, 2005). Essa proposição de mecanismos de apoio ao desenvolvimento do software se estende ao processo de desenvolvimento em si, pois sabe-se que a qualidade do produto está diretamente relacionada com a qualidade do processo empregado.

Apesar de toda a sistematização para o desenvolvimento de software, defeitos podem ainda permanecer no produto desenvolvido. Portanto, atividades de Verificação e Validação são utilizadas durante o desenvolvimento de software com o objetivo de identificar e eliminar defeitos. Dentre essas atividades, o teste de software é uma das mais utilizadas, o qual consiste de uma análise dinâmica do produto e tem por objetivo revelar defeitos podendo, como consequência, aumentar a confiança sobre a qualidade do produto em questão. Nesse sentido, pode-se dizer que um teste bem sucedido é aquele capaz de revelar um defeito ainda não revelado.

A atividade de teste de software é dividida em fases, as quais se relacionam com o próprio processo de desenvolvimento do software, sendo prevista sua aplicação em

três fases (Ammann e Offutt, 2008): *teste de unidade*, em que são testadas as menores unidades que compõem o software; *teste de integração*, em que é testada a comunicação que ocorre entre as unidades que compõem o software; e *teste de sistema*, em que o sistema é testado como um todo, considerando seu relacionamento com outros sistemas analisando propriedades de qualidade esperadas, como por exemplo desempenho, confiabilidade, segurança e funcionalidade.

Um processo de teste adequado envolve atividades de *planejamento*, responsável por definir a maneira com que a atividade de teste será conduzida; *projeto de casos de teste*, responsável pela definição dos casos de teste que serão executados; *execução dos testes*, em que se aplicam os testes elaborados anteriormente; e *análise dos resultados*, responsável por avaliar os resultados do teste, mantendo um levantamento sobre a sua condução (Pressman, 2005; Myers, 2004). Dentre essas atividades, o projeto de casos de teste é considerada a atividade mais importante, pois a qualidade dos testes é diretamente relacionada com os casos de teste utilizados. Outro ponto a destacar é que a própria construção/escolha dos casos de teste pode levar a identificação de defeitos (Myers, 2004).

Idealmente, o programa deveria ser exercitado com todos os valores possíveis do domínio de entrada, entretanto, esse tipo de teste (teste exaustivo) é, em geral, impraticável devido a restrições de tempo e custo para realizá-lo. Assim, para apoiar o projeto de casos de teste, técnicas e critérios de teste podem ser empregados. As técnicas se diferenciam, basicamente, pela origem da informação usada para avaliar ou para gerar os casos de teste, sendo que cada uma possui um conjunto de critérios para esse fim. Um critério é um predicado que é avaliado durante a atividade de teste e que precisa ser satisfeito por um conjunto de casos de teste, sendo dessa forma, usado para guiar o projeto de casos de teste. As principais técnicas de teste são (Myers, 2004):

- **Técnica Funcional**, também conhecida como *teste de caixa preta* pelo fato de tratar o software como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída. Essa técnica utiliza a especificação para derivar os requisitos de teste sendo, portanto, essencial que a especificação esteja correta e represente todos os requisitos do software.

- **Técnica Estrutural**, também conhecida como *teste de caixa branca* pois os requisitos de teste são fortemente baseados na estrutura interna do software, ou seja, nos aspectos de implementação. A maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como *Grafo de Fluxo de Controle (GFC)*, o qual é um grafo orientado onde cada vértice representa um bloco indivisível de comandos do programa e cada aresta representa um desvio de um

bloco para outro. Por meio do GFC escolhem-se os componentes que devem ser executados pelos casos de teste, caracterizando assim o teste estrutural.

- **Técnica Baseada em Defeitos**, a qual utiliza informações sobre os tipos mais comuns de defeitos que podem ser cometidos durante o processo de desenvolvimento de um software para projetar casos de teste (DeMillo, 1987). Um critério conhecido dessa técnica é o teste de mutação, que tem o objetivo de caracterizar um conjunto de casos de teste capaz de indicar que os defeitos não estão presentes no programa em teste. Para isso, defeitos são sistematicamente introduzidos no programa e casos de teste são gerados de modo a indicar que o comportamento do programa em teste está correto comparado ao comportamento do programa com defeitos. Esse critério é fortemente dependente da linguagem de programação, paradigma e artefato testado (especificação, projeto, código), devendo, portanto, estabelecerem-se os defeitos típicos para cada caso antes de se aplicar os critérios dessa técnica.

A utilização de critérios para apoiar a atividade de teste auxiliam a responder duas questões importantes: *1) Qual caso de teste escolher?* e *2) Quando o teste pode ser finalizado?* Uma medida de cobertura dos testes é utilizada para apoiar esse processo, auxiliando a responder a essas questões. A análise de cobertura consiste em determinar o percentual de elementos requeridos por um dado critério de teste que foram exercitados pelo conjunto de casos de teste utilizado. A partir dessa informação, o conjunto de casos de teste pode ser aprimorado acrescentando-se novos casos de teste para exercitar os elementos ainda não cobertos. Nessa perspectiva é fundamental o conhecimento sobre as limitações teóricas relacionadas à atividade de teste, pois os elementos requeridos podem ser não executáveis e, em geral, determinar a não executabilidade de um dado requisito de teste envolve a participação do testador. Ressalta-se também que nenhuma das técnicas de teste é suficiente, quando aplicadas isoladamente, para garantir a qualidade da atividade de teste. As diferentes técnicas se complementam e devem ser aplicadas em conjunto para assegurar um teste de boa qualidade (Maldonado, 1991).

Considerando as diferentes fases, técnicas e critérios de teste, é possível caracterizar testes no nível de especificação e no nível de código. No nível de especificação dois tipos de teste podem ser conduzidos: teste da especificação ou teste baseado na especificação. No teste da especificação o objetivo é similar ao teste de código, buscando garantir que a especificação esteja livre de defeitos e que atenda aos requisitos do usuário. Com esse objetivo, técnicas e critérios de teste definidos para o nível de código foram mapeados e adaptados para o contexto de especificações, principalmente considerando especificações formais (Fabbri et al., 1994, 1995, 1999; Ural et al., 2000; Simao e Maldonado, 2000; Souza et al., 2000b,a, 2001; Sugeta et al., 2004; Briand et al., 2004).

O teste baseado na especificação ou teste baseado em modelos utiliza a especificação como base para testar o programa, analisando se o programa atende os requisitos presentes na especificação. Para esse tipo de teste, é essencial que a especificação esteja correta e retrate os requisitos do usuário, sendo importante, portanto, a condução de testes da especificação. Em geral, esse tipo de teste utiliza modelos que representam a especificação como Máquinas de Estados Finitos, Redes de Petri e Statecharts (Ural, 1992; Petrenko et al., 1996; Simao et al., 2009; Simao e Petrenko, 2010).

A diversidade de domínios de aplicação de software e a proposição de novos paradigmas de desenvolvimento de software requerem que sejam exploradas atividades de teste nesses contextos. Nesse sentido, podem ser identificadas ao longo dos anos várias iniciativas de técnicas e critérios, explorando inicialmente o paradigma convencional de desenvolvimento de software (Herman, 1976; Howden, 1976; DeMillo, 1980; Laski e Korel, 1983; Rapps e Weyuker, 1985; Coward, 1988; Ural e Yang, 1988; Beizer, 1990; Maldonado, 1991), e paradigmas como Orientação a Objetos, Aspectos, dentre outros (Weyuker, 1998; Binder, 2000; Zhao, 2003; Vincenzi et al., 2006; Lemos et al., 2006). Além disso, outra tendência é explorar e adaptar os conceitos de teste, incluindo modelos, técnicas e critérios para domínios específicos de aplicações, como por exemplo, Sistemas de Informação, Computação Distribuída, Sistemas de Tempo Real e Sistemas Embarcados.

Diferentemente dos programas tradicionais, a computação distribuída envolve processos concorrentes que interagem para realizar as tarefas. Essa interação pode ocorrer de forma sincronizada ou não, sendo que esses processos podem ou não concorrer pelos mesmos recursos computacionais. Esse tipo de computação vem sendo cada vez mais empregada e necessária, haja visto as tecnologias atuais, com processadores com múltiplos núcleos e com o uso crescente de *clusters* de computadores. Soluções presentes em telefones móveis, controle de veículos ou sistemas empresariais, onde o acesso remoto a sistemas web corporativos e integrados sob o paradigma SOA (Service-Oriented Architecture) e *cloud computing* fazem uso da concorrência para o seu desenvolvimento ou para permitir a distribuição e replicação de processos e dados.

O teste de aplicações concorrentes torna-se mais complexo, pois além das dificuldades já inerentes à atividade de teste, novos desafios são impostos. Yang e Pollock (Yang e Pollock, 1997) apresentam os principais desafios que estão presentes durante o teste de programas concorrentes:

- Desenvolver novas técnicas de análise estática apropriadas para programas concorrentes;

- Reproduzir uma execução com a mesma entrada de teste e forçar a execução de um caminho na presença de não determinismo;

- Gerar uma representação do programa concorrente que capture informações pertinentes ao teste; 5) investigar a aplicação de critérios de teste seqüenciais para programas concorrentes; e

- Detectar situações não desejadas como: erros de sincronização, comunicação, de fluxo de dados e de *deadlock*[1];

- Projetar critérios de fluxo de dados para programas concorrentes, considerando troca de mensagens e variáveis compartilhadas.

Nesse sentido, várias pesquisas têm explorado a definição de mecanismos de apoio para o teste de programas concorrentes, os quais podem ser divididos em três grupos principais de contribuições:

1. proposição de modelos e critérios de teste (Koppol et al., 2002; Yang e Pollock, 2003; Wong et al., 2005; Lu et al., 2007; Pugh e Ayewah, 2007; Takahashi et al., 2008; Sherman et al., 2009; Krena et al., 2010; Křena et al., 2012; Hong et al., 2012; Tasiran et al., 2012; Hwang et al., 2012; Deng et al., 2013; Gligoric et al., 2010; Sen e Abadir, 2010; Jagannath et al., 2010; Gligoric et al., 2013; Sun et al., 2012; Remenska et al., 2013);

2. proposição de mecanismos de apoio à execução controlada (Ball et al., 2009; Dantas, 2008; Dantas et al., 2007; Emmi et al., 2011; Yu e Narayanasamy, 2009; Kamil e Yelick, 2010; Rungta e Mercer, 2009b; Rungta et al., 2009; Burckhardt et al., 2010; Kundu et al., 2010) e

3. definição de ferramentas de apoio ao teste de programas concorrentes (Yang e Pollock, 2003; Lei e Carver, 2006; Pugh e Ayewah, 2007; Edelstein et al., 2003; Musuvathi et al., 2008; Ball et al., 2010; Nguyen et al., 2008; Pallavi et al., 2009).

Esses trabalhos indicam a relevância da área e os vários aspectos que ainda necessitam de investigação e que motivam novas pesquisas.

O desenvolvimento de estudos experimentais complementa a proposição de novas abordagens de teste, ferramentas e ambientes, permitindo avaliar, dentre outros fatores, a aplicabilidade, escalabilidade, custo e eficácia em revelar defeitos, comparando diferentes abordagens. Para isso, a aplicação de conceitos da Engenharia de Software Experimental é fundamental para que os resultados obtidos sejam válidos e possam ser replicados para novos estudos. A Engenharia de Software Experimental tem por objetivo oferecer mecanismos para apoiar a definição, condução e avaliação de estudos experimentais (Wohlin et al., 2000). Esses mecanismos permitem caracterizar os

[1]Deadlock é uma situação indesejada que pode ocorrer em programas concorrentes, em que um impasse acontece entre dois ou mais processos (*threads*) e esses ficam impedidos de continuar suas execuções, ou seja, ficam bloqueados, um esperando a liberação pelo outro.

objetivos do experimento, definir com clareza suas hipóteses, realizar o experimento adequadamente e analisar os resultados considerando as ameaças à sua validade, aplicando em geral, análise estatística.

Uma das características importantes de um experimento é a possibilidade de sua repetição. Essa repetição permite melhorar o aprendizado dos conceitos investigados e calibrar as características do experimento quando são considerados novos ambientes e sujeitos. Para isso, os dados obtidos devem ser empacotados adequadamente, permitindo a sua repetição e avaliação.

No contexto de teste de software, três fatores principais são usualmente considerados durante a condução de estudos experimentais para a comparação entre critérios: custo, eficácia e dificuldade de satisfação *(strength)* (Mathur e Wong, 1994). *Custo* refere-se ao esforço necessário para que um critério seja usado, o qual pode ser medido pelo número de casos de teste necessários para satisfazer o critério, ou por outras métricas dependentes do critério, tais como: o tempo necessário para executar todos os mutantes gerados ou o tempo gasto para identificar mutantes equivalentes, caminhos e associações não executáveis, construir manualmente os casos de teste e aprender a utilizar as ferramentas de teste. *Eficácia* refere-se à capacidade que um critério possui de detectar um maior número de defeitos em relação a outro critério. *Strength* refere-se à probabilidade de satisfazer-se um critério tendo sido satisfeito um outro. Este fator surge da relação existente entre os critérios de teste onde um critério $C1$ pode incluir um critério $C2$, ou seja, se todo conjunto de casos de teste que cobre os requisitos de $C1$ também cobre os requisitos de $C2$ e o contrário não ocorre pode-se afirmar que $C2$ apresenta um *strength* maior que $C1$. Essa análise é importante porque permite estabelecer na prática uma relação entre os critérios de teste, oferecendo subsídios para a definição de estratégias incrementais de aplicação de critérios de teste.

Os trabalhos desenvolvidos pela autora após a conclusão do doutorado estão inseridos neste contexto de teste de software, com contribuições principais nas linhas de teste de softwares concorrentes e de estudos experimentais em teste de software. As pesquisas desenvolvidas e em desenvolvimento exploram principalmente: 1) mapeamento de critérios de teste para programas concorrentes e especificações formais, 2) definição de mecanismos para a redução do custo da atividade de teste no contexto de programas concorrentes, e 3) avaliação experimental do custo, eficácia em revelar defeitos e aspecto complementar de critérios de teste (*strength*).

1.2 Organização do Texto Sistematizado

Este texto sistematizado apresentada uma descrição das principais contribuições resultantes das atividades de pesquisa realizadas pela autora. No Capítulo 2 são descritos os trabalhos desenvolvidos nas linhas de pesquisa de atuação da autora, dando ênfase

ao relacionamento entre as pesquisas realizadas. Na Seção 2.1 são apresentados os trabalhos relacionados ao teste de programas concorrentes, a qual é a principal linha de atuação da autora. Na Seção 2.2 são apresentados os resultados relacionados ao desenvolvimento de estudos experimentais em teste de software. No Capítulo 3 são discutidas as conclusões a respeito deste texto, indicando os trabalhos futuros e em andamento. No Apêndice A são apresentadas as publicações da autora após a conclusão do doutorado. Por fim, nos Apêndices B a G estão incluídas as publicações mais relevantes, resultantes das pesquisas conduzidas pela autora.

Capítulo

2

Contribuições ao Teste de Software

Este capítulo apresenta uma síntese dos trabalhos desenvolvidos pela autora após o seu doutoramento, incluindo os trabalhos realizados sob sua orientação e os projetos de pesquisa vinculados. Ao longo do texto, as citações referentes a trabalhos em que a autora participa estão em negrito.

As primeiras pesquisas após a conclusão do doutorado (em 2000) foram conduzidas enquanto esta autora ainda era professora da Universidade Estadual de Ponta Grossa (1991 - 2005). Neste período, a universidade não contava com um programa de mestrado e, portanto, as pesquisas foram realizadas com o apoio de alunos de iniciação científica. Em 2001, a autora teve a aprovação de dois projetos de pesquisa no escopo da chamada CNPq Kit-Enxoval Recém-Doutor (processo 68.0101/01-2), intitulados: 1) *Definição de Técnicas e Ferramentas para a Validação de Especificações Formais*; e 2) *Aplicabilidade do Teste de Mutação para a Validação de Aplicações Concorrentes Usando o PVM*. Esses projetos foram desenvolvidos com a cooperação de dois alunos de iniciação científica e resultaram em contribuições em teste de programas concorrentes, gerando duas publicações em periódicos científicos. Os resultados dessas pesquisas estão descritos na Seção 2.1.

Em 2003, a autora teve a aprovação do projeto de pesquisa *ValiPVM: Definição de uma ferramenta para validação de softwares paralelos em ambientes de passagem de mensagens* no escopo da chamada CNPq PDPG-TI (552213/2002-0), cujo objetivo era apoiar e alavancar pequenos grupos de pesquisa e desenvolvimento na área de tecnologia da informação. Esse projeto foi desenvolvido de 2003 a 2005 e contou com a participação de sete alunos de iniciação científica, um aluno de mestrado e com

os docentes: Paulo Sergio Lopes de Souza (na época, docente da UEPG), Silvia Regina Vergílio (UFPR) e Adenilso Simão (ICMC/USP). Esse projeto proporcionou o desenvolvimento de pesquisas sobre teste de programas concorrentes, gerando uma publicação em periódico e quatro publicações em congressos científicos. Os resultados dessas pesquisas são descritos na Seção 2.1.

Como docente do ICMC/USP, a partir de 2005 foi possível começar a orientar trabalhos de mestrado e doutorado junto ao programa de Pós-Graduação em Ciências de Computação e Matemática Computacional do ICMC/USP. Neste ano, deram-se início também as pesquisas sobre o desenvolvimento de estudos experimentais, as quais resultaram em uma publicação em periódico e quatro publicações em congressos científicos. Os resultados dessas pesquisas são descritas na Seção 2.2.

Em 2008, a autora teve a aprovação do projeto de pesquisa *Subsídios para o Teste Estrutural de Aplicações Distribuídas* (FAPESP 2008/04614-5), no qual foram estudados e definidos recursos para reduzir o custo de aplicação do teste de programas concorrentes. Esse projeto contou com a participação dos docentes Paulo Sergio Lopes de Souza (ICMC/USP), Silvia Regina Vergílio (UFPR) e Adenilso Simão (ICMC/USP), contando com uma orientação e uma co-orientação de projetos de mestrado e cinco orientações de iniciação científica. Os principais resultados dessas pesquisas estão descritos na Seção 2.1.

Em 2010 a autora realizou um pós-doutoramento na Universidade de Southampton (Inglaterra), com a colaboração do Prof. Dr. Ed. Zaluska. Durante o pós-doutorado foram realizadas pesquisas sobre teste de programas concorrentes, estendendo alguns resultados anteriores. Os principais resultados são descritos na Seção 2.1.

Dessa forma, as pesquisas realizadas e orientadas pela autora deste texto e suas principais contribuições para a área são apresentadas cronologicamente a seguir, divididas em dois grupos. Inicialmente, na Seção 2.1 são apresentados os trabalhos relacionados ao teste de software concorrente, a qual se caracteriza a linha de atuação principal da autora. Na Seção 2.2 são apresentas as contribuições sobre estudos experimentais em teste de software. Essa linha envolve pequisas sobre avaliação empírica de critérios de teste tanto no contexto de programas sequenciais como de programas concorrentes, contemplando também estudos avaliando a atividade de teste no contexto de ensino de programação.

2.1 Teste de Softwares Concorrentes

Nesta seção são apresentados os resultados obtidos na linha de pesquisa sobre teste de softwares concorrentes, a qual é a linha em que a autora tem atuado mais diretamente após a conclusão de seu doutorado. Inicialmente são apresentados os trabalhos sobre desenvolvimento de modelos, critérios e ferramentas de teste para o contexto de progra-

mas concorrentes. Na Seção 2.1.1 são descritos alguns trabalhos desenvolvidos a partir do doutorado, os quais exploram as mesmas características presentes em programas concorrentes, como não determinismo, comunicação e sincronização, no contexto do teste de especificações formais, os quais se relacionam com essa linha de pesquisa.

Aplicações concorrentes têm sido utilizadas com sucesso para reduzir o tempo computacional em vários domínios. Pode-se citar como exemplo um servidor Web que cria processos (ou *threads*) separados para atender às requisições dos clientes, aumentando com isso o desempenho necessário da aplicação. Em muitos casos, essas aplicações distribuídas utilizam processos concorrentes, quer seja para cooperarem na solução do problema ou por estarem compartilhando recursos em um ambiente comum. Embora a programação concorrente traga inúmeras vantagens para a aplicação desenvolvida, o comportamento não determinístico presente nessas aplicações torna a atividade de teste mais complexa. Múltiplas execuções de um programa concorrente com a mesma entrada podem executar diferentes seqüências de sincronização e podem produzir diferentes resultados. Cabe à atividade de teste, nesse cenário, identificar se todas as seqüências de sincronização possíveis foram executadas e se as saídas obtidas estão corretas. Essa característica difere os programas concorrentes dos programas seqüenciais e precisa ser considerada durante a atividade de teste de software. Portanto, técnicas e critérios de teste específicos para programas concorrentes são necessários, complementando critérios de teste já existentes para programas seqüenciais, pois os aspectos seqüenciais de um programa concorrente também precisam ser considerados.

Alguns trabalhos anteriores apresentam soluções para o teste para programas concorrentes, principalmente focando no paradigma de memória compartilhada (Carver e Tai, 1991; Taylor et al., 1992; Damodaran-Kamal e Francioni, 1993; Yang e Chung, 1992; Krawczyk et al., 1998; Chung et al., 1996; Yang et al., 1998; Koppol et al., 2002; Yang e Pollock, 2003). Esses trabalhos motivaram as pesquisas sobre explorar teste de programas concorrentes para o paradigma de passagem de mensagens.

As pesquisas nessa linha iniciaram no escopo do projeto CNPq Kit-Recém-Doutor 2001 - 2002 (processo 68.0101/01-2), onde foi explorada a definição do teste de mutação para programas concorrentes em PVM (*Parallel Virtual Machine*). O ambiente PVM permite que aplicações paralelas sejam executadas, formadas por processos que se comunicam por meio da troca de mensagens.

O teste de mutação é um critério que vem sendo explorado em diferentes domínios de aplicação, dada a sua fácil adaptação para novos contextos, domínios e fases do desenvolvimento de software. Esse critério utiliza informações sobre os defeitos típicos que podem ser cometidos no processo de desenvolvimento para derivar casos de teste. Assim, os defeitos típicos de um domínio ou paradigma de desenvolvimento são caracterizados e implementados como operadores de mutação que, durante a atividade de teste, geram versões modificadas (mutantes) do produto em teste (especificação ou

implementação, por exemplo). A intenção, com isso, é auxiliar na seleção de casos de teste que demonstrem que os defeitos modelados pelos operadores de mutação não estão presentes no produto em teste. Esse critério depende do conhecimento sobre os defeitos típicos do domínio de aplicação e desse modo, esses defeitos precisam ser caracterizados para que o critério possa ser aplicado.

Existem alguns trabalhos que propõem o teste de mutação para programas concorrentes. Silva-Barradas (Silva-Barradas, 1998) define um conjunto de operadores de mutação para programas concorrentes em Ada. O autor propõe um mecanismo para realizar execução controlada ou determinística, chamado Análise de Mutantes Comportamental. Nesse mecanismo, para cada execução do programa são consideradas a saída obtida e a seqüência de comandos executados, de forma que o comportamento do programa possa ser repetido a partir do valor de entrada e da seqüência percorrida. Outro aspecto importante desse mecanismo é a possibilidade de analisar os mutantes vivos para verificar se existe um caso de teste no conjunto de casos de teste que poderia ter identificado o mutante, mas não o fez devido ao não determinismo. Isso é realizado através de execução simbólica e checagem de modelos. Delamaro et al. (Delamaro et al., 2001) definem um conjunto de operadores de mutação para testar os aspectos de concorrência e sincronização da linguagem Java. Foram identificadas as principais estruturas relacionadas com concorrência e foram definidos operadores de mutação para execução dessas estruturas. Os autores descrevem os operadores de mutação definidos e argumentam que os mesmos cobrem a maioria dos aspectos relacionados à concorrência e sincronização e são também de baixo custo (geram um número pequeno de mutantes). Para tratar o problema de não-determinismo os autores consideram a proposta de Silva-Barradas.

A definição do teste de mutação para aplicações em PVM considera a biblioteca de funções PVM para a linguagem C. Considerando que já existe a definição de operadores de mutação para a linguagem C (Agrawal et al., 1989; Delamaro, 1993), os operadores de mutação definidos concentraram-se essencialmente nas características relacionadas ao paralelismo e à comunicação dessas aplicações. O conjunto de operadores de mutação foi definido tendo como objetivo torná-lo o mais completo possível, englobando as principais características que podem ser encontradas em programas em PVM. Os operadores de mutação definidos procuram modelar os seguintes tipos de erros: 1) erro no fluxo de dados entre as tarefas (ou processos); 2) erro no empacotamento de mensagens; e 3) erro na sincronização e no paralelismo das tarefas. Com base nesses possíveis tipos de erros, 15 operadores de mutação foram definidos. Em **(Giacometti et al., 2002)** os operadores são apresentados, juntamente com os resultados de sua aplicação em programas clássicos implementados em PVM. Este trabalho foi desenvolvido com a cooperação do aluno de iniciação científica Cassiano Giacometti, com bolsa projeto CNPq Kit-Enxoval Recém-Doutor (68.0101/01-2).

Entre 2003 e 2005 a autora coordenou o projeto *ValiPVM: Definição de uma ferramenta para validação de softwares paralelos em ambientes de passagem de mensagens*, aprovado no escopo da chamada CNPq PDPG-TI (552213/2002-0). Esse projeto foi desenvolvido inicialmente em cooperação com a Professora Dra. Silvia Regina Vergilio (UFPR), Professor Dr. Paulo Sergio Lopes de Souza (UEPG, na época) e Professor Dr. Adenilso Simão (ICMC). O objetivo central do projeto era explorar a definição de critérios estruturais para programas concorrentes em PVM e MPI.

O teste estrutural, ou teste caixa branca, utiliza a estrutura interna do programa para apoiar a definição de critérios de teste. Em geral, a estrutura do programa é representada na forma de um grafo de fluxo de controle (GFC), o qual é um modelo de teste em que cada nó representa um bloco de comandos sem desvio de controle (ou seja, todos os comandos de um bloco são executados se esse bloco é percorrido por um teste), e cada aresta representa o desvio de controle entre dois blocos.

O modelo CFG foi estendido de modo a acomodar as características presentes em programas concorrentes. Inicialmente, foi definido um modelo de teste para programas concorrentes que se comunicam via troca de mensagens. Esse modelo, denominado grafo de fluxo de controle paralelo (PCFG), é formado pelos GFCs de cada processo concorrente, somado com as arestas que representam a comunicação entre esses processos. Ou seja, se um nó n_1 do GFC de um processo p^i possui um comando de envio de mensagem que pode ser recebida por um nó n_2 do GFC de um processo p^j que possui um comando de recebimento de mensagem, então uma aresta inter-processo entre os nós n_1 e n_2 é criada **(Vergilio et al., 2005; Souza et al., 2008b)**.

A partir do PCFG, uma família de critérios de teste foi definida, a qual é composta por critérios que testam aspectos sequenciais, a comunicação e a sincronização presente nas aplicações concorrentes, considerando o paradigma de passagem de mensagens **(Vergilio et al., 2005; Souza et al., 2008b)**. Esses critérios foram definidos com base nos critérios estruturais de Rapps e Weyuker (Rapps e Weyuker, 1985) e consideram o fluxo de controle, o fluxo de dados e o fluxo de comunicação presentes no programa concorrente. Em relação ao fluxo de controle, são estabelecidos critérios que exercitam arestas, nós e caminhos no GFCP. Em relação ao fluxo de dados e de comunicação, o objetivo é exercitar os possíveis pares entre definição e uso de variáveis do programa. Definição ocorre toda vez que um valor é atribuído a uma variável, enquanto que o uso ocorre toda vez que existe uma referência a uma variável, podendo ocorrer em uma computação, conhecido como *uso computacional*, e em comandos de repetição ou decisão, conhecido como *uso predicativo*. Em Vergilio et al. **(Vergilio et al., 2005)** um novo conceito de uso de variável para o contexto de programas concorrentes foi definido, o qual considera o uso de variáveis em primitivas de comunicação, ou seja, variáveis usadas para a troca de mensagens entre os processos da aplicação.

Esses critérios de teste contribuem com o processo de teste de aplicações concorrentes, fornecendo uma medida de cobertura sobre o processo de teste, a qual pode ser utilizada para medir a qualidade da atividade de teste. Os critérios de teste foram inspirados nos critérios existentes para programas seqüenciais, de modo que aspectos seqüenciais do programa concorrente possam ser testados. Além disso, a proposta não é baseada em um modelo de defeitos e sim em aumentar a cobertura dos testes realizados e apresentar elementos que precisam ser testados. Desse modo, defeitos que porventura existam podem ser revelados durante o processo de teste **(Souza et al., 2008b)**.

Com base no modelo de teste e critérios, a ferramenta de teste ValiPar foi definida. Essa ferramenta foi inicialmente instanciada para PVM (ValiPVM) **(Souza et al., 2005, 2008a)** e MPI (ValiMPI) **(Hausen et al., 2006, 2007)**. Essas contribuições contaram com a participação dos seguintes alunos de iniciação científica, financiados com bolsas do projeto CNPq PDPG-TI (552213/2002-0): José Reinaldo Perim Filho, Thiago B. Gonçalves, Alexandre de Melo Lima, João Walter Bruno Filho, Luciana V. B. Wiecheteck e Luiz Duda. Os resultados dessas pesquisas estão divulgadas no livro *Introdução ao Teste de Software*, no capítulo *Teste de Programas Concorrentes* **(Souza et al., 2007b)**.

Dando continuidade à implementação da ferramenta ValiPar, o trabalho do aluno de iniciação científica Matheus Lin T. Alvarenga (bolsa CNPq PIBIC) contribuiu com a melhoria do módulo de instrumentação da ferramenta. A instrumentação é um passo da ferramenta de teste em que o programa é transformado em uma versão onde são adicionadas informações para geração de um trace da execução. Essas informações são relevantes para avaliar a cobertura dos testes. Para a instrumentação de programas concorrentes, foi necessário adequar a ferramenta *IDeL - Instrumentation Description Language* (Simao et al., 2002), pois a mesma possuia versão apenas para a instrumentação de programas na linguagem C. Neste trabalho, o instrumentador foi estudado e estendido para considerar novas primitivas de comunicação presentes em programas MPI.

No trabalho do aluno de iniciação científica Laércio C. Asano (bolsa FAPESP 2010/05237-0) foi investigada a geração automática de dados de teste para o contexto de programas concorrentes. Para isso, o trabalho desenvolvido por Ferreira e Vergilio (Ferreira e Vergilio, 2005) foi adaptado para esse contexto. No trabalho de Ferreira e Vergilio, foram utilizados algoritmos genéticos evolutivos que são capazes de gerar novos dados de teste com base em informações que precisam ser cobertas durante os testes. Esse algoritmo funciona juntamente com a ferramenta de teste Poketool (Chaim, 1991), a qual apóia o teste estrutural de programas em C. Durante a iniciação científica, o algoritmo foi estudado e adaptado para os critérios propostos para programas concorrentes. Com algumas limitações, esse algoritmo foi capaz de gerar dados de teste

para programas concorrentes em MPI. As limitações estão sendo investigadas em um projeto de mestrado em andamento, em desenvolvimento pelo aluno José Dario Pintor (bolsa CNPq).

O modelo de teste e critérios definidos para o contexto de passagem de mensagem foram estendidos para contemplar o paradigma de memória compartilhada **(Sarmanho et al., 2007)**. Nesse paradigma, a comunicação entre as threads ocorre por meio de variáveis compartilhadas, sendo que essa comunicação é implícita, ou seja, toda vez que uma thread deseja se comunicar com outra, ela deve escrever os dados na variável compartilhada e em seguida a outra thread deve ler o conteúdo da variável, estabelecendo-se a comunicação. O desafio nesse caso é estabelecer mecanismos para identificar pares de comunicação e estabelecer os requisitos de teste. Os requisitos são as informações mínimas que precisam ser cobertas pelos casos de teste. Por exemplo, um requisito poderia ser executar todo uso de uma determinada variável do programa. Devido ao não determinismo, os pares de comunicação são estabelecidos em tempo de execução, de maneira dinâmica, e os requisitos de teste devem ser determinados antes da execução, de maneira estática. Para solucionar isso foi empregado o conceito de *timestamp*, oriundo do teste de alcançabilidade (Lei e Carver, 2006). O teste de alcançabilidade é uma abordagem dinâmica de teste que executa diferentes sequências de sincronização entre processos sem construir um modelo estático. A partir do resultado de uma execução do programa, o teste de alcançabilidade gera automaticamente todas as outras sincronizações que podem ocorrer. Para isso, o programa é executado deterministicamente até certo ponto e, a partir desse ponto a execução é feita de forma não-determinística, de modo a permitir que novas sincronizações possíveis sejam geradas. Apesar do custo envolvido, essa abordagem tem a vantagem de gerar somente sincronizações executáveis, o que é um grande benefício durante os testes. Dessa forma, o conceito de *timestamp* foi empregado de modo a estabelecer uma ordem entre os eventos concorrentes do programa e em seguida obter os pares de comunicação exercitados em uma execução. As idéias propostas foram implementadas em uma ferramenta de apoio, denominada ValiPThread, que segue a mesma arquitetura definida pela ValiPar, adaptando os módulos necessários **(Sarmanho et al., 2008)**. Este trabalho de mestrado foi desenvolvido pelo aluno Felipe Sarmanho, orientado do Professor Dr. Paulo Sergio Lopes de Souza, onde a autora atuou como colaboradora do trabalho.

O modelo de teste definido para programas concorrentes mostrou-se bastante adequado para representar sistemas no contexto de composição de serviços web. Desse modo, em Endo et al. **(Endo et al., 2007)**, o modelo foi mapeado para o contexto de Web services, onde os critérios de teste foram adaptados. Uma ferramenta de teste, denominada ValiBPEL foi implementada, a qual apóia o teste de aplicações em BPEL **(Endo et al., 2008)**. Observou-se que em algumas situações esse modelo de teste baseado em GFC não é suficiente para representar adequadamente todos os as-

pectos de uma composição de serviços. Assim, em Endo et al. **(Endo et al., 2010)** foi proposta uma estratégia que combina duas abordagens: a abordagem baseada em cobertura, como as dos trabalhos citados acima, e a abordagem baseada em eventos, proposta por Belli et al. (Belli et al., 2006).

Entre 2008 e 2010 a autora coordenou o projeto *Subsídios para o Teste Estrutural de Aplicações Distribuídas*, financiado pela FAPESP (processo n.2008/04614-5). O objetivo deste projeto era identificar e propor alternativas para reduzir o custo de aplicação da atividade de teste no contexto de programas concorrentes. Para dar apoio ao desenvolvimento deste projeto de pesquisa uma revisão sistemática foi desenvolvida.

Revisão sistemática é um processo de avaliação e interpretação de todos os estudos disponíveis sobre uma questão de pesquisa particular, fornecendo um conhecimento para futuras pesquisas (Kitchenham, 2004). A revisão sistemática desenvolvida em **(Brito et al., 2010a)** teve o objetivo de caracterizar o estado da arte no que tange a estudos experimentais, classificação de defeitos e critérios de teste para programas concorrentes. Para o levantamento, foram consideradas as principais bibliotecas digitais da área, incluindo *ACM Digital Library, IEEE eXplore, Engineering Village e SCOPUS*, considerando todas as contribuições até 2009. Um total de 44 trabalhos foram selecionados e classificados de acordo com suas contribuições na área de teste de programas concorrentes. Os resultados indicaram uma falta de trabalhos sobre estudos experimentais avaliando critérios e ferramentas de teste para programas concorrentes. Os resultados dessa revisão motivaram um convite para palestra de abertura no *PADTAD*[1]. Para a palestra, a revisão sistemática foi atualizada e reclassificada considerando três contribuições essenciais: 1) abordagens de teste, 2) classificação de defeitos e 3) ferramenta de teste. Como resultado, observou-se que a maioria das contribuições focam em programas com memória compartilhada, possivelmente incentivados pelo interesse em pesquisas sobre programas concorrentes em Java, desenvolvidas principalmente pela IBM e Microsoft **(Souza et al., 2011a)**.

Um dos problemas do teste estrutural aplicado a programas concorrentes é o alto custo dos testes, devido ao grande número de requisitos de teste que são gerados e, por consequência, o elevado número de requisitos não executáveis. Não executabilidade é considerada uma limitação da atividade de teste, pois além do custo envolvido, não existe mecanismo que automatize sua determinação. Desse modo, no trabalho de mestrado de Mario Cesar Machado, orientado do Professor Dr. Paulo Sergio Lopes de Souza com a colaboração da autora deste texto, foram investigados mecanismos para reduzir esse custo de aplicação. Dentre as contribuições propostas estão a definição de heurísticas para identificação e eliminação de alguns requisitos não executáveis e a definição de uma abordagem de teste que usa o teste de alcançabilidade para guiar

[1] *IX Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, co-alocado ao *ISSTA - International Symposium on Software Testing and Analysis*, Toronto, Junho, 2011

a execução dos testes. A abordagem definida usa as informações sobre elementos que precisam ser cobertos pelos casos de teste para guiar a escolha de novas sincronizações a serem executadas pela árvore de alcançabilidade. A ideia é aproveitar as vantagens de ambas as abordagens, sendo que os requisitos dos critérios de teste permitem reduzir o custo de aplicação do teste de alcançabilidade, mantendo ainda uma medida de cobertura e, o teste de alcançabilidade oferece a vantagem de gerar somente sincronizações executáveis, o que é uma informação relevante para o teste estrutural **(Souza et al., 2011b)**. Um artigo com avaliação experimental desses resultados encontra-se aceito **(Souza et al., 2014b)**.

Em outro estudo, o aluno Mário Santos Camillo, em seu trabalho de conclusão de curso, avaliou o impacto da aplicação de testes temporais durante os testes de programas concorrentes. Como os critérios baseados em sincronizações em geral produzem muitos requisitos de teste, muitos dos quais não são executáveis, é importante que se determine formas de se identificar quais sincronizações não podem ocorrer durante a execução do programa. Assim, em uma iniciação científica anterior (orientação do Professor Dr. Adenilso Simão), o aluno investigou como a estratégia proposta por Lei e Carver (Lei e Carver, 2006) poderia ser utilizada para gerar somente as sincronizações executáveis. Nesse trabalho de conclusão, o aluno analisou empiricamente essa abordagem, obtendo resultados que indicam que o teste temporal pode diminuir o esforço do testador durante o projeto de testes.

A avaliação experimental de critérios de teste é uma atividade relevante pois permite determinar a aplicabilidade, aspecto complementar e eficácia em revelar defeito dos mesmos, contribuindo com o estabelecimento de uma estratégia viável de aplicação dos critérios propostos.

Nessa direção, o trabalho de mestrado da aluna Maria Adelina Silva Brito (bolsa FAPESP 2009/04517-2) foi desenvolvido com o objetivo de definir e aplicar um estudo experimental para avaliar a efetividade dos critérios de teste definidos para programas concorrentes no contexto de ambientes de passagem de mensagens **(Brito e Souza, 2010)**. Esse estudo foi desenvolvido em sintonia com os pontos básicos de comparação teórica e empírica de critérios de teste empregados em outros trabalhos, como custo de aplicação, eficácia em revelar defeitos e aspecto complementar (*strength*) entre critérios de teste (Mathur e Wong, 1994; Wong et al., 1994). Para desenvolvimento do experimento, o processo de Engenharia de Software Experimental proposto por Wohlin foi empregado (Wohlin et al., 2000). Os resultados indicaram que os critérios para programas concorrentes apresentam diferentes custos de aplicação e que são capazes de revelar diferentes tipos de defeitos, indicando que é interessante empregar diferentes tipos de critérios durante a validação de aplicações nesse domínio **(Brito et al., 2013)**.

De maneira similar, o trabalho de mestrado de Silvana Morita Melo (bolsa FAPESP 2010/04042-1) teve como objetivo avaliar experimentalmente os critérios de teste pro-

postos para programas concorrentes com memória compartilhada **(Melo et al., 2012)**. As mesmas ideias exploradas em Brito et al. **(Brito e Souza, 2010)** foram empregadas para definição do experimento. Entretanto, como diferencial, esse experimento utilizou benchmarks de programas consolidados já utilizados pela comunidade de programação concorrente, como Inspect (Yang et al., 2008), Helgrind (Valgrind-Developers, 2011) e Rungta (Rungta e Mercer, 2009a). Nesse experimento foi analisado também o diferencial em termos de custo e eficácia em revelar defeitos quando comparados os critérios específicos para programas concorrentes e os critérios que analisam só o aspecto sequencial dos programas. Os resultados indicaram que existem diferentes custos de aplicação entre os critérios e que os critérios específicos para o teste da concorrência são mais caros. Observou-se também que alguns tipos de defeitos só são revelados por critérios específicos, o que motiva a proposição de critérios que abordem os aspectos específicos do domínio de aplicação.

Conforme comentado anteriormente, o teste de mutação é um critério que tem apresentado uma alta eficácia em revelar defeitos. Além disso, essa técnica tem sido amplamente utilizada para avaliar a eficácia em revelar defeitos de outras técnicas de validação. Para isso, o teste de mutação é utilizado para gerar versões do programa contendo defeitos e então experimentos são conduzidos para avaliar o quanto que uma técnica ou critério de teste é capaz de revelar os defeitos modelados pelos operadores de mutação. O uso do teste de mutação nesse contexto é vantajoso pois esse critério apresenta um processo bem definido e repetível de semear os defeitos no produto a ser avaliado. Isso permite facilmente a repetição de estudos em outros experimentos (Andrews et al., 2005).

A definição do teste de mutação para o contexto de programas concorrentes é desafiadora, pois é preciso considerar que o não determinismo possibilitará a ocorrência de mais de uma saída diferente e correta para o programa. Uma vez que isso acontece, o fato da execução do mutante gerar um resultado diferente do resultado gerado pela execução do programa original não é o suficiente para distinguir o mutante, pois a discrepância dos resultados pode ter ocorrido pela ação do não determinismo e não por causa da mutação implementada.

A ideia principal do teste de mutação é que um mutante executado com um caso de teste t deve ser morto caso seu comportamento seja diferente do comportamento do programa P do qual o mutante foi derivado. Em programas concorrentes seria necessário comparar o conjunto de todos os resultados possíveis de um mutante M quando executado com um caso de teste t em relação ao conjunto de todos os resultados possíveis do programa original P com o caso de teste t. Se M apresentar um comportamento diferente, ele foi distinguido de P. Entretanto, em geral, não é possível conhecer todo o comportamento possível de um programa concorrente para determinada entrada, pois é um processo caro.

Essas questões foram investigadas durante o desenvolvimento do trabalho de mestrado de Rodolfo Adamshuk Silva (bolsa FAPESP 2010/04935-6). Neste trabalho foram identificados os defeitos típicos que são cometidos no contexto de programas concorrentes em MPI e, a partir desses defeitos, um conjunto de operadores de mutação foi proposto **(Silva et al., 2012b)**. Esses operadores englobam mutações em todas as funções de comunicação empregadas no padrão MPI, focando principalmente em defeitos relacionados à comunicação e à sincronização entre os processos da aplicação. Neste trabalho foram propostas também abordagens de execução determinística de mutantes, de modo a evitar os problemas advindos do comportamento não determinístico de programas concorrentes. Nesse sentido, duas abordagens foram propostas. A primeira abordagem utiliza o teste de alcançabilidade (Lei e Carver, 2006) para gerar todas as possíveis sequências de sincronização para um caso de teste executado e posteriormente forçar o mutante a executar cada uma dessas sequências de sincronização e avaliar a saída. Se o mutante apresentar uma saída diferente em relação ao programa original ou não conseguir executar a sequência, ele é considerado morto. A segunda abordagem torna-se mais flexível, pois não força o mutante a percorrer uma determinada sequência de sincronização. Nessa abordagem o teste de alcançabilidade é utilizado para gerar todas as possíveis sequências de sincronização para um caso de teste executado. O mutante é executado também pelo caso de teste e a sincronização obtida pelo mutante é comparada com o conjunto de sincronizações obtido pelo programa original. Um mutante é considerado como morto se ele apresentar um resultado diferente do obtido pelo programa original **(Silva et al., 2012a)**. Uma ferramenta de apoio, denominada ValiMut, implementa os operadores de mutação definidos e as abordagens de execução de mutantes para programas concorrentes **(Silva et al., 2013)**. O trabalho de conclusão de curso do aluno Henrique Mancebo Russo deu apoio à implementação dos operadores de mutação.

O modelo de teste para programas concorrentes com passagem de mensagens proposto anteriormente **(Souza et al., 2008b)** não contemplava algumas primitivas de comunicação existentes, como por exemplo, as primitivas ponto-a-ponto e coletivas não bloqueantes. Essas primitivas são comumente empregadas em programas concorrentes com passagem de mensagens. Apesar de que programas contendo essas primitivas pudessem ser testados pelo modelo anterior, esse fato gerava um custo maior para o teste. Desse modo, em Souza et al. **(Souza et al., 2014a)** o modelo de teste proposto em **(Souza et al., 2008b)** é revisitado e uma nova versão é proposta, a qual engloba um conjunto maior de primitivas de comunicação e uma nova sistemática na geração de requisitos de teste que permite reduzir o número de elementos que precisam ser testados. Uma das melhorias propostas é reconhecer durante a análise estática quais são as possíveis sincronizações entre *send* e *receive*, de modo que sejam geradas somente as possíveis, diminuindo assim o número de elementos não executáveis gerados. Com-

parado com o modelo anterior, foi possível reduzir em até 93% o número de elementos não executáveis para critérios relacionados a arestas de sincronização entre processos.

Existem algumas aplicações concorrentes que podem empregar ambos os paradigmas de comunicação, passagem de mensagens e memória compartilhada. Para esses casos, os modelos de teste definidos anteriormente não se enquadram, pois atendem programas com paradigmas específicos. Nessa direção, no trabalho de Souza et al. **(Souza et al., 2013)** é proposto um novo modelo de teste que considera tanto passagem de mensagem como memória compartilhada, focando principalmente na análise de fluxo de dados que ocorre entre os processos e *threads*. Nesse modelo, o programa é formado por uma hierarquia de tarefas concorrentes, podendo conter n processos e cada processo podendo conter m *threads*. A comunicação e sincronização pode ser entre *threads* de um mesmo processo ou de processos diferentes. O PCFG é composto de CFGs para cada *thread* e pelas diferentes arestas de comunicação que podem ocorrer, ou seja, arestas que representam troca de mensagens entre processos (arestas inter-processos) e arestas que representam compartilhamento de variáveis entre *threads* (arestas inter-threads). O modelo foi validado considerando aplicações concorrentes na linguagem Java e os resultados indicaram que o modelo é capaz de extrair as informações de fluxo de dados e de comunicação que são pertinentes à atividade de teste de programas concorrentes.

2.1.1 Teste de Especificação de Sistemas Reativos

Dando continuidade às pesquisas conduzidas durante o doutorado, existem algumas contribuições relacionadas ao teste de especificação de sistemas reativos. Sistemas reativos são aqueles que reagem a estímulos internos ou do ambiente, de forma a produzir resultados corretos dentro de intervalos de tempo previamente especificados. Esses sistemas, em geral, apresentam um comportamento bastante complexo e, portanto, técnicas formais são indicadas para a sua modelagem, destacando-se o uso de técnicas de descrição formais, como Estelle, SDL e Lotos (Turner, 1993), como baseada em modelos, como Statecharts e Máquinas de Estados Finito (MEFs) (Harel et al., 1987; Gill, 1962).

A exploração de critérios de teste para especificação surgiu da necessidade de garantir especificações corretas. Sabe-se que mesmo utilizando técnicas formais não é possível garantir que a especificação esteja livre de defeitos e que a mesma satisfaça todos os requisitos do usuário, pois devido à interferência humana, defeitos podem ser inseridos nessa fase de desenvolvimento. Pesquisas anteriores do grupo de Engenharia de Software do ICMC propuseram o teste de mutação aplicado à especificação em Statecharts, Redes de Petri e MEFs (Fabbri et al., 1994, 1995, 1999). Durante o doutorado da autora foram definidos e analisados teoricamente critérios de teste (estruturais e de

mutação) para especificações em Estelle e em Statecharts **(Souza et al., 2000a,b, 2001)**.

Nesse contexto, foi definida e implementada a geração da árvore de alcançabilidade para especificações em Estelle **(Zambianco et al., 2002)**, a qual permitiu aplicar os critérios estruturais definidos em Souza et al. **(Souza et al., 2001)**. A árvore de alcançabilidade consiste em gerar o comportamento possível do sistema modelado, representando todos os estados alcançáveis quando considerados todos os eventos que podem ocorrer. Essa representação é de interesse pois é possível então avaliar o comportamento da especificação frente a diferentes combinações de eventos. Este trabalho foi desenvolvido no escopo do projeto CNPq Kit-Recém-Doutor 2001 - 2002 (processo 68.0101/01-2) com o apoio do aluno de iniciação científica Rivaldo Zambianco Junior (bolsa CNPq).

Na sequência, houve a continuação do desenvolvimento da ferramenta de teste *SCCTool - Specification Coverage Criteria Testing Tool*, de apoio à aplicação dos critérios estruturais para Statecharts **(Souza et al., 2000a)**. Desse modo, com o apoio da aluna de iniciação científica Bianca Pitarello, o módulo de avaliação de seqüências de teste foi implementado estendendo assim as funcionalidades da ferramenta SCCTool **(Pitarello e Souza, 2006)**. Essa ferramenta foi inserida no contexto de um ambiente para apoio ao teste de especificação de sistemas espaciais, desenvolvido pelo Instituto de Pesquisas Espaciais - INPE (Santiago et al., 2008, 2012), a partir de colaborações iniciadas dentro do escopo do projeto *Plavis - Plataforma para Validação e Integração de Software em Sistemas Espaciais*, coordenado pelo Prof. Dr. José Carlos Maldonado (Projeto CNPq/2003).

Na sequência, os critérios estruturais definidos para Statecharts e Estelle foram mapeados para o contexto de especificações em Redes de Petri. Este trabalho foi desenvolvido em colaboração com o Prof. Dr. Adenilso Simão **(Simao et al., 2003)**.

Outra cooperação nessa linha de pesquisa é a participação no projeto desenvolvido pelo aluno de mestrado Jorge Francisco Cutigi, orientado do Prof. Dr. Adenilso Simão. Neste trabalho, o aluno investigou um método de redução de custos para geração de sequências de teste para especificações modeladas em Máquinas de Estados Finitos. Os resultados indicaram uma redução expressiva no número de sequências de teste geradas **(Cutigi et al., 2010)**.

Essas pesquisas iniciais motivaram a investigação do teste de software aplicado a programas concorrentes, pois durante esses estudos explorou-se o aspecto comportamental de sistemas reativos, analisando a comunicação, sincronização e comportamento não determinístico desses sistemas no nível da especificação. Essas características estão presentes também em aplicações concorrentes e, portanto, muito do conhecimento gerado nessas pesquisas foi aproveitado e mapeado do nível de especificação para o nível de programa.

Resumo

Os trabalhos nessa linha contaram com a colaboração de vários alunos de mestrado e iniciação científica. Contaram também com a colaboração de diversos pesquisadores, em especial, Paulo Sergio Lopes de Souza(ICMC), Silvia Regina Vergilio (UFPR), Adenilso Simão (ICMC) e Ed Zaluska (Universidade de Southampton), com quem realizou o pós-doutoramento. Foram orientados três trabalhos de mestrado e 16 trabalhos de iniciação científica. Conforme será descrito no Capítulo 3, encontram-se em desenvolvimento três trabalhos de doutorado e dois trabalhos de mestrado vinculados a essa linha de pesquisa. Foram publicados dois artigos em revista internacional, dois artigos em revista nacional, um capítulo de livro, 22 artigos em eventos internacionais e 8 artigos em eventos nacionais.

Vinculados a essa linha de pesquisa foram aprovados quatro projetos de pesquisa. Além disso, houve a participação da autora em outros projetos de pesquisa, nas quais essas pesquisas foram de interesse e permitiram promover novas cooperações. Em especial, destaca-se a participação da autora nos seguintes projetos de pesquisa:

- Projeto *Teste paralelo de programas concorrentes*, coordenado pelo Prof. Dr. Paulo Sergio Lopes de Souza, com financiamento da FAPESP, a partir de 2013, em andamento, contando com a colaboração de pesquisadores internacionais;

- *Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos (INCT/SEC)*, coordenado pelo Prof. Dr. José Carlos Maldonado, com financiamento do CNPq (573963/2008-8) e da FAPESP (08/57870-9), a partir de 2009, em andamento;

- Projeto CNPq Universal *Subsídios para atividades de VV&T no desenvolvimento de software*, coordenado pelo Prof. Dr. José Carlos Maldonado, de 2008 a 2011;

- Projeto Procad de Cooperação Acadêmica (ICMC/UEM/PUC-RS): *Integrando e Aprimorando Atividades de Pesquisa, Ensino/Treinamento e Transferência Tecnológica em Teste e Validação de Software*, coordenado pelo Prof. Dr. José Carlos Maldonado, de 2008 a 2012;

- Projeto *Plavis - Plataforma para Validação e Integração de Software em Sistemas Espaciais*, coordenado pelo Prof. Dr. José Carlos Maldonado, com financiamento do CNPq, de 2003 a 2006.

2.2 Experimentação em Teste de Software

O estudo sobre técnicas, critérios e ferramentas de teste tem se tornando uma preocupação constante de pesquisadores da área, buscando avaliar as diferentes propostas existentes, de modo a gerar resultados sobre comparações e aplicações das abordagens. Em 2000 Harrold (Harrold, 2000) apresentou um relato contendo uma visão das principais contribuições para a área de teste de software até aquele ano e preconizou como uma das direções futuras o desenvolvimento de experimentos comparativos entre técnicas. Seus argumentos mostravam que já existia uma grande quantidade de propostas de técnicas e critérios de teste e, dessa forma, mais importante que propor novas, deveriam ser estudadas cuidadosamente as vantagens de cada uma, frente às suas limitações e custos envolvidos. De fato, esse tipo de resultado é necessário para transferir para a indústria as técnicas, critérios e metodologias propostas pela academia.

A Experimentação em Engenharia de Software tem como objetivo caracterizar, avaliar, prever, controlar ou melhorar tanto os produtos, como também os processos, recursos, modelos ou teorias. Wohlin et al (Wohlin et al., 2000) afirmam que a experimentação pode proporcionar uma base de conhecimento para reduzir incertezas sobre quais teorias, ferramentas e metodologias são adequadas, como também descobrir novas áreas de pesquisa ou conduzir as teorias para direções promissoras. Por meio de experimentos é possível avaliar um objeto de estudo (técnica, método, etc.) por várias pessoas e em vários ambientes diferentes. Quanto mais um experimento for replicado em ambientes e culturas diferentes, mais dados se obtêm sobre o objeto de estudo, o que faz com que sua caracterização seja mais significativa. A replicação de um experimento depende do suporte de um Pacote de Laboratório, o qual deve conter toda informação necessária para que o experimento seja realizado da forma mais fiel possível em relação ao experimento original.

Dependendo do propósito da avaliação e das condições necessárias para a investigação empírica, três tipos de investigação podem ser conduzidos: *survey*, estudo de caso e experimento controlado. No *Survey* são realizadas investigações em retrospectiva, ou seja, são conduzidas para avaliar uma técnica ou ferramenta que se encontra em uso por algum tempo. Estudo de caso é usado para monitorar o andamento de um projeto ou atividade. Em geral, o estudo de caso visa avaliar um elemento específico e é definido como um estudo observacional. Experimento controlado é caracterizado pelo controle sistemático das variáveis e do processo, tendo como objetivos confirmar teorias, conhecimento convencional, explorar os relacionamentos, avaliar a predição dos modelos ou validar medidas. Além disso, o experimento controlado envolve a formulação de hipóteses, que precisam ser verificadas em relação aos resultados obtidos.

Segundo Wohlin et al (Wohlin et al., 2000) a experimentação caracteriza-se por um processo que tem início a partir da definição do experimento passando pelo pla-

nejamento, operação, análise e interpretação, até a apresentação e empacotamento do mesmo. Durante a apresentação e o empacotamento do experimento há várias questões que devem ser consideradas, dentre elas, registrar todas as informações usadas, geradas e as condições nas quais o experimento foi executado, de modo que, em uma nova aplicação, os resultados possam ser comparados de uma maneira efetiva. Conforme apontado por Basili (Basili, 1996), o empacotamento padronizado dos dados experimentais pode servir como base para a criação de bibliotecas de experimentação, de modo que uma base de conhecimento possa ser derivada e utilizada posteriormente.

Considerando a diversidade de critérios que têm sido estabelecidos e reconhecido o caráter complementar das técnicas e critérios de teste (Mathur, 1991; Mathur e Wong, 1993, 1994; Wong et al., 1994), um ponto crucial é a escolha e/ou a determinação de uma estratégia de teste, que em última análise passa pela escolha de critérios de teste, de forma que as vantagens de cada um desses critérios sejam combinadas objetivando uma atividade de teste de maior qualidade. Estudos teóricos e empíricos de critérios de teste são de extrema relevância para a formação desse conhecimento, fornecendo subsídios para o estabelecimento de estratégias de baixo custo e alta eficácia. Diversos esforços da comunidade científica nessa direção são identificados. Uma das primeiras contribuições nessa linha de pesquisa foi a sumarização dessas contribuições, dando origem a um capítulo de livro, intitulado *Estudos Teóricos e Experimentais* do livro *Introdução ao Teste de Software* **(Souza et al., 2007a)**.

No trabalho de mestrado de Marllos Paiva Prado foi realizado um estudo comparativo avaliando o custo e o aspecto complementar de critérios de teste nos paradigmas procedimental e orientado a objetos (OO). O objetivo foi avaliar qual é o impacto do paradigma para a atividade de teste em termos de custo e esforço, considerando o teste estrutural. Seguindo o processo de engenharia de software experimental, o experimento foi caracterizado de modo a garantir sua repetibilidade **(Prado et al., 2008)**. Para o estudo, um conjunto composto de 32 programas do domínio de estruturas de dados foi utilizado, sendo que tais programas foram extraídos dos livros de Ziviani (Ziviani, 2005b,a). Apesar do paradigma OO possibilitar o desenvolvimento de códigos menores quando comparados ao paradigma procedimental, os resultados obtidos indicaram que não existe de fato uma diferença de custo e de aspecto complementar devido ao paradigma de programação empregado **(Prado et al., 2010; Souza et al., 2012)**.

O trabalho de mestrado de Diogo N. Campanha complementou o trabalho de Prado **(Prado et al., 2008)**. Neste trabalho, o estudo envolveu os mesmos programas e resultados já obtidos por Prado, comparando os paradigmas em relação ao critério análise de mutantes **(Campanha et al., 2010)**. Diferentemente dos resultados obtidos pelo experimento de Prado **(Prado et al., 2008)**, os resultados indicam que tanto o custo quanto o aspecto complementar do teste de mutação é maior em programas implementados no paradigma procedimental do que no paradigma OO. Pelo fato das

funções no paradigma procedimental serem maiores que suas equivalentes em OO, o número de mutantes gerados é maior e com isso aumenta-se o custo e o esforço dos testes.

Algumas pesquisas têm explorado o desenvolvimento de estudos experimentais no contexto do ensino de teste de software e de programação. Nessa direção, em Barbosa et al. **(Barbosa et al., 2007)** são descritos os resultados da replicação de um experimento durante o ensino de disciplinas de graduação na área de teste de software. O experimento replicado foi aplicado anteriormente por Maldonado et al. (Maldonado et al., 2006) e visa comparar as atividades de teste e inspeção de software. Na replicação feita procurou-se avaliar a efetividade de aprendizado alcançado em função dos resultados obtidos. De modo geral observou-se que o aprendizado em relação à aplicação das técnicas de inspeção e teste foi satisfatório tanto no que tange a geração de casos de teste como o grau de cobertura alcançado e a quantidade de defeitos identificados pelos alunos. Em **(Barbosa et al., 2008)** esse estudo é analisando do ponto de vista do material de aprendizagem empregado, o qual foi desenvolvido seguindo a abordagem IMA-CID (Barbosa e Maldonado, 2006) para o contexto dessa disciplina de teste. Os resultados indicam que o material conseguiu apoiar o desenvolvimento do experimento, auxiliando os estudantes em aplicar corretamente as técnicas e critérios utilizados.

No experimento descrito em Campanha et al. **(Campanha et al., 2009)** foi avaliada a reutilização de conjuntos de teste gerados pelo critério análise de mutantes em programas equivalentes. Esse artigo analisa empiricamente os resultados teóricos obtidos por Weyuker (Weyuker, 1986), onde ela afirma que um conjunto de casos de teste (T) adequado para o critério Análise de Mutantes (AM) – AM-adequado – para um programa P não é adequado para um programa Q, equivalente a P. Um conjunto de casos de teste é adequado quando ele é capaz de cobrir todos os elementos requeridos do programa. Apesar de que teoricamente isso não é possível, sabe-se que esse conjunto pode ser próximo ao adequado, motivando a sua reutilização. Em um trabalho anterior **(Maldonado et al., 1995)**, verificaram-se indícios de que a adequação é de fato similar quando se utilizam conjuntos de teste de referência aplicados a outros programas que implementam o mesmo problema. Entretanto, nesse primeiro estudo foi utilizada apenas uma implementação de cada algoritmo de ordenação, desenvolvida pelos próprios autores. No experimento de Campanha et al. **(Campanha et al., 2009)** foram utilizados 6 diferentes algoritmos de ordenação contra 22 implementações diferentes construídas por alunos de graduação. Para verificar a similaridade das médias das coberturas atingidas pelos diferentes conjuntos de teste, foram realizados testes estatísticos de análise de variância. Os resultados indicaram que não há diferença significativa entre as coberturas atingidas pelos diferentes conjuntos, o que motiva a sua reutilização, auxiliando na redução do esforço na geração de dados de teste.

No experimento descrito em Brito et al. **(Brito et al., 2010b, 2012)** a ideia de reutilização de conjuntos de teste também foi explorada, entretanto focando em seu apoio no ensino de disciplinas introdutórias de programação. A ideia foi investigar o apoio de introduzir testes durante o ensino de programação como alternativa para aumento na qualidade dos programas gerados pelos alunos. O estudo foi realizado no domínio de manipulação de vetores e matrizes. Foram gerados conjuntos de casos de teste funcional com base em programas de referência. Esses conjuntos de teste foram reutilizados pelos alunos para o teste de seus programas, os quais eram equivalentes aos programas de referência. Os resultados indicaram que a qualidade dos programas desenvolvidos pelos alunos, tendo esse conjunto para teste, melhorou significativamente quando comparado com o grupo de alunos que testou os programas conforme seu próprio entendimento de testes. Isso motiva a introdução de conceitos de teste e o uso de ferramentas de apoio durante o ensino de disciplinas introdutórias de programação.

No experimento descrito em Felizardo et al **(Felizardo et al., 2013)** foi realizada a replicação de um experimento anterior, o qual avalia os benefícios do uso de VTM - *Visual Text Mining* para apoiar a seleção de estudos primários em revisões sistemáticas. VTM é uma técnica que associa algoritmos de mineração de dados e técnicas de visualização para apoiar a exploração de dados. No contexto de revisões sistemáticas, essa abordagem pode auxiliar na identificação de estudos primários relevantes (Felizardo et al., 2010). Assim, o experimento conduzido em (Felizardo et al., 2011) foi re-executado no contexto de uma disciplina de pós-graduação (SSC5903-Revisão Sistemática em Engenharia de Software) avaliando-se o tempo e a corretitude da seleção de estudos primários utilizando VTM contra a seleção manual. Os resultados obtidos confirmaram os resultados anteriores, indicando que VTM pode aumentar o tempo de seleção de estudos, aumentando também a quantidade de estudos corretamente incluídos/excluídos.

No contexto de teste de programas concorrentes, estudos experimentais têm sido conduzidos e foram descritos na Seção 2.1, os quais têm contribuído com a avaliação de custo, eficácia em revelar defeitos e aspecto complementar de critérios de teste **(Brito e Souza, 2010; Melo et al., 2012; Brito et al., 2013)**. Essas comparações são fundamentais para o estabelecimento de estratégias de aplicação de modo a aproveitar as vantagens de cada técnica e critério de teste. Além disso, esses estudos favorecem a disponibilização para a indústria dos mecanismos definidos na academia.

Resumo

Os trabalhos nessa linha contaram com a colaboração de alunos de mestrado e uma aluna de pós-doutorado. Contou também com diversos pesquisadores, em especial, Ellen Francine Barbosa (ICMC) e José Carlos Maldonado (ICMC). Foram orientados dois trabalhos de mestrado. Foram publicados dois artigos em revista internacional,

um capítulo de livro, seis artigos em eventos internacionais e dois artigos em eventos nacionais.

2.3 Considerações Finais

Neste capítulo foram apresentadas as principais publicações da autora após a realização do seu doutorado, juntamente com os projetos aprovados e participação em projetos de pesquisa.

Na Figura 2.1 as publicações são apresentadas separadas por ano e considerando as contribuições em teste de programas concorrentes e em experimentação, seguindo a seguinte representação: círculos para publicações em periódicos e quadrados para publicações em eventos. As seis publicações mais relevantes são destacas em cinza. Três publicações (7, 32 e 34) se referem aos dois temas de pesquisa. A numeração corresponde às referências listadas no Apêndice A.

É possível observar que as contribuições se intensificaram a partir de 2005, principalmente em função de publicar resultados de projetos iniciados em 2003 e também por iniciar suas atividades no ICMC, onde a autora pode começar a orientar na Pós-Graduação. Com isso, iniciaram-se as publicações sobre Experimentação e intensificaram-se as publicações em Teste de Programas Concorrentes.

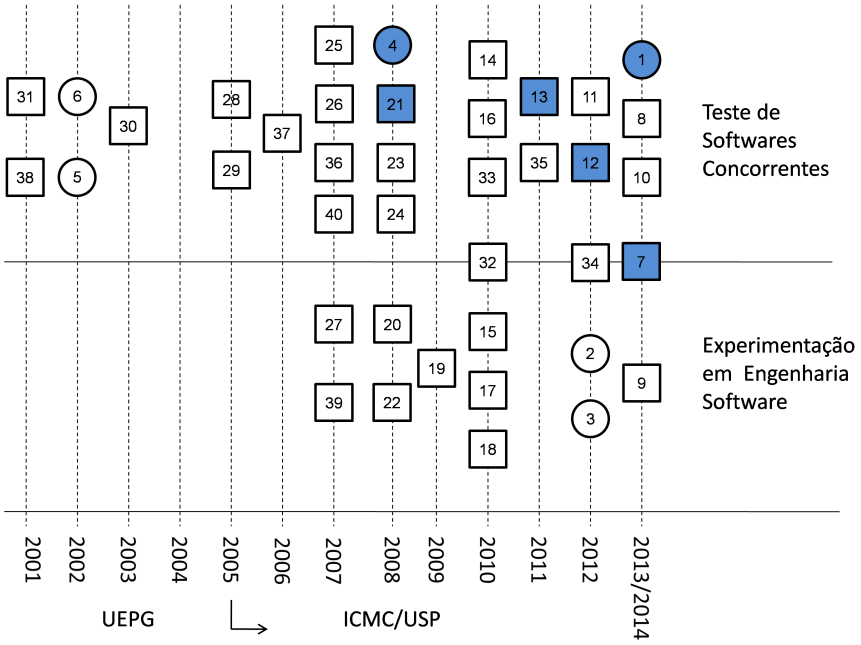


Figura 2.1: Distribuição das publicações por linha de pesquisa

As pesquisas concentram-se na atividade de teste de software, explorando as diferentes técnicas de teste: funcional (teste de especificação), estrutural (em programas concorrentes e em experimentação) e baseado em defeitos (teste de mutação em progra-

mas concorrentes e experimentação). Além disso, explorou-se essa atividade aplicada a diferentes domínios de aplicação, como sistemas reativos e distribuídos, apresentando com isso um caráter multidisciplinar.

Conforme apontado por Maldonado (Maldonado, 1991) e Harrold (Harrold, 2000) as contribuições na área de teste de software podem ser subdividas em três categorias:

1. Estudos teóricos: estudos que permitem o avanço do conhecimento na área, propondo soluções para os problemas existentes, investigando e apresentando limitações das soluções existentes.

2. Estudos experimentais: estudos que permitem a comparação entre abordagens de teste já existentes, indicando na prática sua aplicabilidade, custo, eficácia, dentre outros fatores de interesse.

3. Automatização: envolve o desenvolvimento de ferramentas para apoiar a aplicação de abordagens de teste definidas.

Considerando essas categorias de contribuições, a Figura 2.2 apresenta como as publicações da autora podem ser distribuídas nessas categorias. Pode-se observar que existem contribuições nas três categorias, com um maior enfoque nos estudos teóricos e experimentais. Alguns dos estudos experimentais são relacionados a avaliações empíricas de estudos teóricos conduzidos pela autora, compreendendo uma avaliação empírica de abordagens propostas teoricamente.

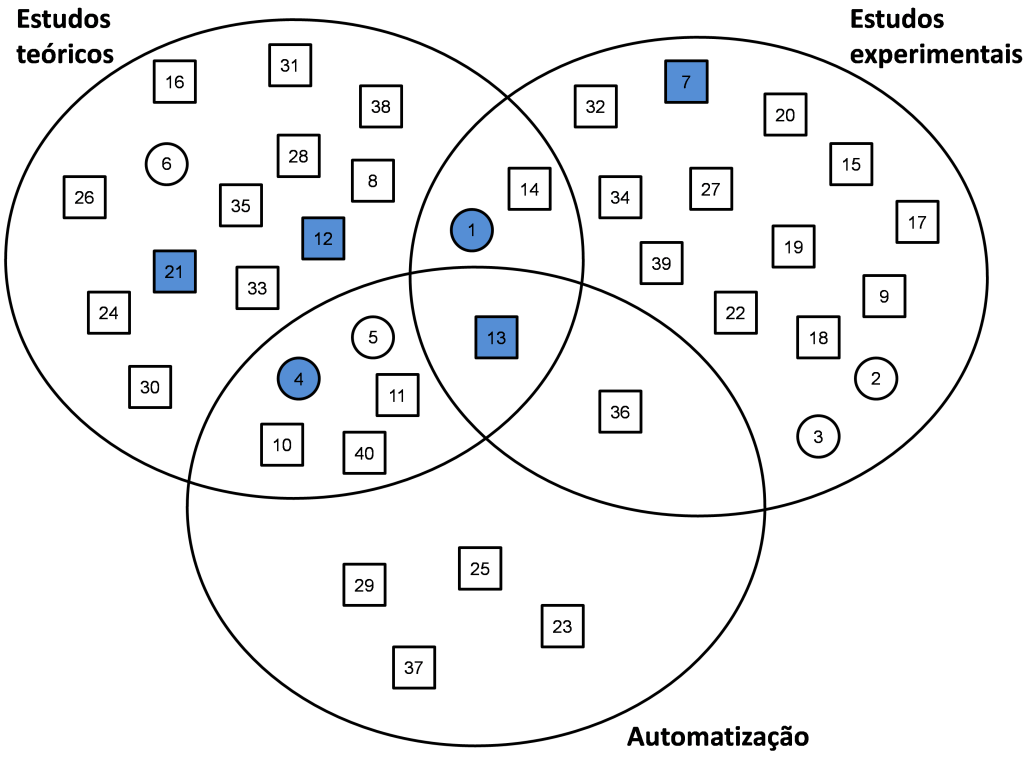


Figura 2.2: Distribuição das publicações pelas áreas de contribuição em teste de software

Capítulo

3

Conclusões

3.1 Discussões e Reflexões

Este texto apresenta as contribuições da autora após a conclusão do seu doutorado. Suas contribuições concentram-se na área de teste de software, com maior ênfase em pesquisas sobre teste de programas concorrentes e experimentação em teste de software.

Teste de programas concorrentes é a linha principal de pesquisa da autora, com diversas contribuições teóricas, experimentais e de automatização. Aplicar testes nesse contexto é desafiador, pois novos problemas precisam ser explorados, principalmente relacionados ao comportamento não determinístico. Assim, novos modelos de teste foram definidos e critérios de teste estruturais e de mutação foram propostos. Ferramentas de apoio foram desenvolvidas de modo a validar os modelos e critérios definidos. Contemplando essas contribuições, estudos experimentais foram desenvolvidos buscando comparar e avaliar as abordagens propostas, de modo a identificar limitações e propor estratégias de aplicação. Revisões sistemáticas foram conduzidas as quais proporcionaram identificar *gaps* na área, norteando novos trabalhos e desdobramentos.

Experimentação em teste de software é uma linha de atuação importante para a autora, na qual foram conduzidas investigações desde o seu mestrado. As contribuições em experimentação apresentadas neste documento se relacionam a diferentes contextos, destacando-se estudos comparativos de critérios de teste tanto para programas sequenciais como para programas concorrentes e estudos experimentais aplicados ao ensino de programação e teste de software. Esses estudos fornecem importantes resultados que

podem direcionar novas pesquisas e melhorar atividades de ensino de teste de software e de disciplinas introdutórias de programação.

A obtenção de recursos por meio de projetos de pesquisa proporcionou o desenvolvimento dos trabalhos aqui descritos. Foram aprovados 5 projetos de pesquisa, com financiamentos para equipamentos, bolsas de iniciação científica e participação em eventos. Os dois primeiros do CNPq na chamada kit-recém doutor, o terceiro do CNPq na chamada PDPG-TI, o quarto projeto regular da FAPESP e o quinto da Capes, projeto de estágio no exterior, o qual proporcionou o desenvolvimento do pós-doutorado. O terceiro projeto (CNPq PDPG-TI 552213/2002-0) se destaca, pois proporcionou a criação de um laboratório de pesquisa em Engenharia de Software na UEPG, universidade onde a autora estava vinculada em 2003, ano que o projeto foi aprovado. Isso permitiu criar e estabelecer um grupo de pesquisa em engenharia de software nessa universidade, a qual não contava com programa de mestrado na área de computação até aquele momento. Nesse período após a conclusão do doutorado, houve também a participação da autora em 7 projetos de pesquisa.

Apenas os trabalhos desenvolvidos e finalizados após a conclusão do doutorado foram apresentados neste documento. Existem outros trabalhos em desenvolvimento, os quais são descritos na próxima seção.

Atualmente, a autora possui uma boa inserção no cenário acadêmico. A autora é membro do comitê de programas de diversos eventos importantes na sua área de atuação, como *SBES - Simpósio Brasileiro de Engenharia de Software*, *ESELAW - Experimental Latin American Workshop*, *Mutation - International Workshop on Mutation Analysis*, *SAST - Brazilian Workshop On Systematic and Automated Software Testing*, *PADTAD - Workshop on Parallel and Distributed Systems* e *SBSI - Simpósio Brasileiro de Sistemas de Informação*. Tem atuado com revisora de artigos em periódicos importantes da área, como: *Software Testing, Verification and Reliability*, *Information and Software Technology* e *Science of Computer Programming*. Foi também co-chair de um evento internacional - *10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging - PADTAD 2012* e de um evento nacional - *III Brazilian Workshop On Systematic and Automated Software Testing SAST2009*.

3.2 Trabalhos em Andamento e Futuros

Atualmente, a autora orienta três alunos de doutorado e quatro alunos de mestrado, todos com temas relacionados a testes e/ou experimentação. Os três alunos de doutorado foram alunos de mestrado da autora, o que proporciona a continuidade na condução das pesquisas e na formação dos alunos.

No trabalho de doutorado da aluna **Maria Adelina Silva Brito** (bolsa FAPESP 2013/02321-9), investiga-se a definição de testes para sistemas embarcados críticos,

dando continuidade às pesquisas realizadas no mestrado da aluna. Este projeto é relacionado ao INCT/SEC e visa propor mecanismos de teste no nível de integração de software para sistemas robóticos móveis. Os softwares integrados nesses sistemas apresentam características similares aos encontrados em programas concorrentes, como comunicação e sincronização. Com isso, tem-se a intenção de explorar nesse domínio as soluções propostas para o teste estrutural de programas concorrentes, entretanto, considerando a comunicação entre os softwares desses sistemas.

No trabalho de doutorado da aluna **Silvana Morita Melo** (bolsa FAPESP 2013/ 05046-9), investiga-se a proposição de um framework para apoiar a avaliação de técnicas e ferramentas de teste no contexto de programação concorrente, dando continuidade às pesquisas realizadas no mestrado da aluna. O objetivo é coletar evidências sobre potenciais benefícios, limitações, custo e riscos associados as abordagens propostas, oferecendo informações para que a indústria possa extrair vantagens dessas abordagens para a validação de suas aplicações.

No trabalho de doutorado do aluno **Rodolfo Adamshuk Silva** (bolsa CNPq), investiga-se a utilização de técnicas de *Search-Based Software Testing* para otimizar o teste de mutação de programas concorrentes, dando continuidade às pesquisas realizadas no mestrado do aluno. *Search-Based Software Testing* é uma técnica que busca soluções eficientes para problemas na área de teste de software, por meio da aplicação de algoritmos genéticos (metaheurísticas) (McMinn, 2004). Um dos problemas que se pretende investigar é o alto custo relacionado ao número de mutantes que são gerados por esse teste. Sabe-se que alguns mutantes são mais eficientes que outros e desse modo pretende-se usar esses algoritmos para identificar quais são os melhores locais do programa para realizar as mutações, de modo a reduzir o número de mutantes gerados, porém realizando a mutação em pontos críticos do programa concorrente.

No trabalho de mestrado do aluno **Jose Dario Pintor da Silva** (bolsa CNPq), investiga-se a proposição de uma abordagem para geração automática de dados de teste para programas concorrentes, capaz de auxiliar na identificação de defeitos comuns relacionados com essas aplicações, como *deadlock*, *starvation* e erros de sincronização. Algoritmos oriundos de *Search-Based Software Testing* também estão sendo explorados no escopo deste trabalho, buscando-se uma técnica efetiva de geração de dados de teste.

O trabalho de mestrado do aluno **Marcos Pereira dos Santos** (bolsa CNPq) está relacionado às pesquisas da aluna de doutorado Maria Adelina Silva Brito e visa investigar a adaptação dos critérios de teste propostos para programas concorrentes para o contexto de sistemas embarcados robóticos. Para isso, o aluno irá conduzir estudos comparativos avaliando esses critérios em relação aos ambientes utilizados para o desenvolvimento de sistemas robóticos. O estudo comparativo deverá considerar, dentre outros fatores, a dificuldade de realização dos testes nesses ambientes, facilidade de aplicação dos testes em estudos de casos que utilizam os ambientes, adaptabilidade

dos critérios de teste de programas concorrentes para estes ambientes de desenvolvimento e a efetividade dos critérios para esse contexto de aplicação. Esses estudos estão considerando as aplicações reais desenvolvidas no âmbito do INCT/SEC.

No trabalho de mestrado do aluno **Bruno Henrique Oliveira** (bolsa CNPq), investiga-se a aplicação de conceitos da engenharia de software experimental, avaliando qualidade de software em ambientes de desenvolvimento com métodos ágeis. Sabe-se que as métricas de qualidade propostas para processos tradicionais de desenvolvimento de software não se adere facilmente ao processo ágil, fato que motivou essa investigação. A ideia é realizar estudos de caso e *surveys* de modo a coletar o estado da prática em relação à avaliação de qualidade dos produtos de software e propor métricas que podem ser empregadas para esse fim quando empresas empregam métodos ágeis.

No trabalho de mestrado do aluno **Ricardo Fontão Verhaeg** (bolsa CNPq), investigam-se técnicas para a redução do esforço na geração e aplicação da atividade de teste de software no contexto de métodos ágeis. Esse trabalho está relacionado com o mestrado do aluno Bruno Henrique Oliveira. A partir dos resultados obtidos por Bruno e do estudo das técnicas existentes para redução de esforço, este trabalho pretende propor uma abordagem de apoio para o desenvolvimento de testes no contexto ágil, de modo a minimizar os custos e aumentar a qualidade dos testes nesse contexto.

Além desses trabalhos em andamento, a autora participa efetivamente no desenvolvimento de outros trabalhos de mestrado e doutorado relacionados ao teste de programas concorrentes, tais como:

- Investigação de teste de programas concorrentes aplicado ao contexto de linguagens funcionais, como *Erlang*, a qual apresenta problemas similares encontrados em programas concorrentes, como não determinismo. Este trabalho está sendo desenvolvido pelo aluno de doutorado Alexandre Ponce de Oliveira, sob orientação do Prof. Paulo Sergio Lopes de Souza;

- Definição de mecanismos de paralelização aplicados ao teste de programas concorrentes, em desenvolvimento pelo aluno de mestrado Raphael Negrisoli Batista, sob orientação do Prof. Paulo Sergio Lopes de Souza;

- Definição de um ambiente de aplicação de teste de programas concorrentes orientado a serviços, em desenvolvimento pelo aluno de mestrado Rafael Regis Prado, sob orientação do Prof. Paulo Sergio Lopes de Souza;

- Identificação e desenvolvimento de benchmarks de programas para apoio ao teste de programas concorrentes, considerando diferentes complexidades e características de concorrência, necessárias para a validação de abordagens propostas para programas concorrentes. Embora a área possua alguns benchmarks disponíveis,

os mesmos possuem algoritmos concorrentes mais simples do ponto de vista da comunicação e sincronização. Desse modo, os benchmarks propostos irão permitir, dentre outras características: uso simultâneo dos dois paradigmas de comunicação e sincronização, aumento gradativo do grau de complexidade das iterações realizadas entre os processos e documentação adequada sobre os programas concorrentes. Este trabalho está em desenvolvimento pelo aluno de mestrado George Gabriel Mendes Dourado, sob orientação do Prof. Paulo Sergio Lopes de Souza.

Atualmente, a autora participa ativamente do projeto FAPESP *Teste Paralelo de Programas Concorrentes*, coordenado pelo Prof. Paulo Sergio Lopes de Souza, o qual investiga a paralelização de atividades do teste de programas concorrentes. Este projeto está sendo desenvolvido em cooperação com os pesquisadores internacionais Prof. Ed. Zaluska (Universidade de Southampton), com o qual se estabeleceu cooperação no período do Pós-Doutorado, e com o Prof. João Lourenço (Universidade Nova de Lisboa), colaboração iniciada durante a organização em conjunto do Workshop PADTAD 2012 [1]. Este projeto de pesquisa aborda o problema do custo, propondo a paralelização da atividade de teste estrutural de programas concorrentes desenvolvidos nos paradigmas de passagem de mensagens e memória compartilhada. O objetivo é a redução do tempo de resposta do teste, reduzindo-se com isso o custo sem reduzir a qualidade dessa atividade.

Além das pesquisas em desenvolvimento descritas acima, podem-se destacar como trabalhos futuros, os seguintes estudos:

- Explorar teste de especificação no contexto de programas concorrentes, visando a adaptação de modelos para a validação de programas concorrentes;
- Estudo e definição de novos mecanismos para melhorar o teste de programas concorrentes, por meio de priorização e de estratégias de apoio para a localização de defeitos, com base em trabalhos relacionados a esses assuntos;
- Definir e desenvolver novos estudos experimentais no contexto do teste de programas concorrentes, visando avaliar e comparar as propostas em desenvolvimento na área de geração automática de dados; de novos modelos e critérios de teste; e de redução de custos do teste;
- Definir e desenvolver novos estudos experimentais ou replicar estudos já aplicados no contexto de teste de software, considerando o contexto de ensino, tanto de teste de software, como de disciplinas introdutórias de programação;
- Investigar a aplicação do teste de mutação para o apoio ao teste de desempenho de sistemas. Esse trabalho é fruto de uma parceria iniciada no escopo do projeto

[1] http://faculty.uoit.ca/bradbury/padtad2012/

Procad de Cooperação Acadêmica (ICMC/UEM/PUC-RS), junto com docentes e alunos da PUC-RS, os quais têm desenvolvido ambientes de apoio ao teste de desempenho. Relacionado a esse tema, um artigo em periódico encontra-se aceito, o qual foi elaborado em parceria com o grupo da PUC-RS **(Rodrigues et al., 2014)**.

Agradecimentos

Aos meus alunos e ex-alunos da graduação e da pós-graduação pela convivência, pelo aprendizado e principalmente pelas contribuições que permitiram a condução de todos os meus projetos de pesquisa. Em especial, agradeço à Maria, Rodolfo e Silvana por toda a ajuda durante a preparação do meu memorial.

Aos pesquisadores do grupo de Engenharia de Software do ICMC/USP, incluindo alunos e professores, pela amizade e apoio no desenvolvimento de minhas pesquisas, desde a época da minha pós-graduação.

Aos professores José Carlos Maldonado, Paulo Sergio Lopes de Souza, Silvia Regina Vergilio, Adenilso Simão, Ellen Francine Barbosa e Alexandre Delbem pela colaboração nessas pesquisas e em outros projetos.

Ao SSC e demais comunidade do ICMC/USP pela estrutura de trabalho, que direta ou indiretamente apoiaram minhas pesquisas. Em especial, agradeço à Glauciema pela pronta ajuda em todos os momentos necessários.

À FAPESP, ao CNPq e à CAPES pelo apoio financeiro aos projetos e aos alunos.

Aos meus pais e irmãos que sempre me apoiaram e entenderam minhas ausências.

Aos meus filhos Felipe Augusto e Ana Rafaela pelo amor, carinho e enorme paciência. Ao Paulo pelo amor, dedicação, companheirismo e incentivo constantes. Agradeço a Deus por vocês fazerem parte da minha vida e das minhas conquistas.

A todos os meus amigos que sabem que é para eles que estou agradecendo pela amizade, pela convivência, pelos ensinamentos e pelo apoio constante. A vocês minha gratidão e consideração!

A Deus pelas oportunidades concedidas, pela saúde, por tudo.

Referências

Agrawal, H.; DeMillo, R.; Hathaway, R.; Hsu, W.; Hsu, W.; Krauser, E.; Martin, R. J.; Mathur, A. *Design of mutant operators for the c programming language.* Relatório Técnico SERC-TR-41-P, Software Eng. Research Center, Purdue University, 1989.

Ammann, P.; Offutt, J. *Introduction to software testing.* Cambridge University Press, 2008.

Andrews, J.; Briand, L.; Labiche, Y. Is mutation an appropriate tool for testing experiments? In: *International Conference on Software Engineering (ICSE/´05)*, St. Louis, Missouri, USA., 2005, p. 15–21.

Ball, T.; Burckhardt, S.; Coons, K. E.; Musuvathi, M.; Qadeer, S. *Preemption sealing for efficient concurrency testing.* Relatório Técnico, Microsoft, 2009.

Ball, T.; Burckhardt, S.; Halleux, P.; Musuvathi, M.; Qadeer, S. Predictable and progressive testing of multi-threaded code. *IEEE Software*, v. 99, n. PrePrints, p. 75–83, 2010.

Barbosa, E. F.; Maldonado, J. C. An integrated content modeling approach for educational modules. In: *IFIP 19th World Computer Congress – International Conference on Education for the 21st Century*, Santiago, Chile, 2006, p. 17–26.

Barbosa, E. F.; Souza, S. R. S.; Domingues, A. L. S.; Chan, A.; Nina, E.; Maldonado, J. C. Uma experiência no ensino de inspeção e teste de software. In: *VI Simpósio Brasileiro de Qualidade de Software (SBQS)*, Porto de Galinhas, Brazil, 2007, p. 309–324.

Barbosa, E. F.; Souza, S. R. S.; Maldonado, J. C. An experience on applying learning mechanisms for teaching inspection and software testing. In: *21st Conference on Software Engineering Education & Training*, Charleston, South Carolina, 2008, p. 189–196.

Basili, V. R. The Role of Experimentation in Software Engineering: Past, Current, and Future. In: *ICSE*, 1996, p. 442–449.

Beizer, B. *Software system testing.* Van Nostrand Reinhold Company, 1990.

Belli, F.; Budnik, C. J.; White, L. Event-based modelling, analysis and testing of user interactions: approach and case study: Research articles. *Software Testing, Verification & Reliability*, v. 16, n. 1, p. 3–32, 2006.

Binder, R. V. *Testing object-oriented systems.* Addison-Wesley, 2000.

Briand, L. C.; Labiche, Y.; Wang, Y. *Using simulation to empirically investigate test coverage criteria based on statechart.* Relatório Técnico SCE-02-09, Carleton University, 2004.

Brito, M. A. S.; Felizardo, K.; Souza, P. S. L.; Souza, S. R. S. Concurrent software testing: A systematic review. In: *22nd IFIP International Conference on Testing Software and Systems*, Natal, Brazil, 2010a, p. 79–84.

Brito, M. A. S.; Rossi, J.; Braga, R. T. V.; Souza, S. R. S. An experience on applying software testing for teaching introductory programming courses. *CLEI Electronic Journal*, v. 15, n. 4, p. 1–12, 2012.

Brito, M. A. S.; Rossi, J.; Souza, S. R. S.; Braga, R. T. V. Reuso de conjuntos de casos de teste no ensino de disciplinas introdutórias de programação. In: *ESELAW'2010 - VII Experimental Software Engineering Latin American Workshop*, Goiania, 2010b, p. 101–110.

Brito, M. A. S.; Souza, S. R. S. Avaliação da efetividade dos critérios de teste estruturais no contexto de programas concorrentes. In: *15º Workshop de Teses e Dissertações em Engenharia de Software (WTES'2010), 24º Simpósio Brasileiro de Engenharia de Software (SBES'2010), Congresso Brasileiro de Software: Teoria e Prática (CBSoft'2010)*, 2010, p. 31–36.

Brito, M. A. S.; Souza, S. R. S.; Souza, P. S. L. An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs. In: *ICCS - International Conference on Computational Science*, 2013, p. 250–259.

Burckhardt, S.; Kothari, P.; Musuvathi, M.; Nagarakatte, S. A randomized scheduler with probabilistic guarantees of finding bugs. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, 2010, p. 167–178.

Campanha, D. N.; Souza, S. R. S.; Lemos, O. A. L.; Barbosa, E. F.; Maldonado, J. C. Reutilização de conjuntos de teste: Um estudo no domínio de algoritmos de ordenação. In: *ESELAW'2009 - VI Experimental Software Engineering Latin American Workshop. Brazilian Computer Society - SBC*, São Carlos, 2009, p. 114 – 123.

Campanha, D. N.; Souza, S. R. S.; Maldonado, J. C. Mutation testing in procedural and object-oriented paradigms: An evaluation of data structure programs. In: *XXIV Simpósio Brasileiro de Engenharia de Software(SBES'2010), Congresso Brasileiro de Software: Teoria e Prática (CBSoft'2010)*, Salvador, 2010, p. 91–100.

Carver, R.; Tai, K.-C. Replay and testing for concurrent programs. *IEEE Software*, p. 74–86, 1991.

Chaim, M. J. *Poketool - uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados.* Dissertação de Mestrado, DCA/FEE/UNICAMP - Campinas, SP, 1991.

Chung, C.-M.; Shih, T. K.; Wang, Y.-H.; Lin, W.-C.; Kou, Y.-F. Task decomposition testing and metrics for concurrent programs. In: *Fifth International Symposium on Software Reliability Engineering (ISSRE'96)*, 1996, p. 122–130.

Coward, P. A review of software testing. *Information and Software Technology*, v. 30, n. 3, p. 189–198, 1988.

Cutigi, J. F.; Ribeiro, P. H.; Simao, A.; Souza, S. R. S. Redução do número de sequências no teste de conformidade de protocolos. In: *XI Workshop de Testes e Tolerância a Falhas*, Gramado, Brazil, 2010, p. 105–117.

Damodaran-Kamal, S. K.; Francioni, J. M. Nondeterminacy: Testing and debugging in message passing parallel programs. In: *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, New York: ACM Press, 1993, p. 118–128.

Dantas, A. Improving developers' confidence in test results of multi-threaded systems: avoiding early and late assertions. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, 2008, p. 899–900.

Dantas, A.; Gaudencio, M.; Brasileiro, F.; Cirne, W. *Obtaining trustworthy test results in multi-threaded systems.* Relatório Técnico, Federal University of Campina Grande, 2007.

Delamaro, M. E. *Proteum - um ambiente de teste baseado na análise de mutantes*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (ICMC/USP), 1993.

Delamaro, M. E.; Pezzè, M.; Vincenzi, A. M.; Maldonado, J. C. Mutant operators for testing concurrent java programs. In: *XV SBES - Simpósio Brasileiro de Engenharia de Software*, 2001, p. 272–285.

DeMillo, R. A. Mutation analysis as a tool for software quality assurance. In: *Proceedings of COMPSAC80, Chicago - IL*, 1980.

DeMillo, R. A. *Software testing and evaluation*. The Benjamin/Cummings Publishing Company, 1987.

Deng, D.; Zhang, W.; Lu, S. Efficient concurrency-bug detection across inputs. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, New York, NY, USA: ACM, 2013, p. 785–802 (*OOPSLA '13*,).

Edelstein, O.; Farchi, E.; Goldin, E.; Nir, Y.; Ratsaby, G.; Ur, S. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, v. 15, n. 3-5, p. 485–499, 2003.

Emmi, M.; Qadeer, S.; RakamariA, Z. Delay-bounded scheduling. *ACM SIGPLAN Notices (POPL 2011)*, v. 46, n. 1, p. 411–422, 2011.

Endo, A. T.; Lindshulte, M.; Simao, A. S.; Souza, S. R. S. Event and coverage-based testing of web services. In: *2nd Workshop on Model-Based Verification & Validation from Research to Practice (MVV) - in conjunction with the 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, Cingapura, 2010, p. 1–8.

Endo, A. T.; Simao, A. S.; Souza, S. R. S.; Souza, P. S. L. Aplicação de teste estrutural para composição de web services. In: *SAST2007 - 1st Brazilian Workshop on Systematic and Automated Software Testing - XXI SBES - Simpósio Brasileiro de Engenharia de Software*, João Pessoa, PB, 2007, p. 13–20.

Endo, A. T.; Simao, A. S.; Souza, S. R. S.; Souza, P. S. L. Web services composition testing: A strategy based on structural testing of parallel programs. In: *TaicPart: Testing Academic & Industrial Conference - Practice and Research Techniques*, Windsor, 2008, p. 3–12.

Fabbri, S. C. P. F.; Maldonado, J. C.; Delamaro, M. E.; Masiero, P. C. Mutation analysis testing for finite state machine. In: *Fifth International Symposium on Software Reliability Engineering (ISSRE'94)*, 1994, p. 220–229.

Fabbri, S. C. P. F.; Maldonado, J. C.; Masiero, P. C.; Delamaro, M. E. Mutation analysis applied to validate specifications based on petri nets. In: *8th IFIP Conference on Formal Descriptions Techniques for Distributed Systems and Communication Protocol (FORTE'95)*, Montreal, Canadá, 1995.

Fabbri, S. C. P. F.; Maldonado, J. C.; Sugeta, T.; Masiero, P. C. Mutation testing applied to validate specifications based on statecharts. In: Society, I. C., ed. *International Symposium on Software Reliability Engineering (ISSRE'99)*, Florida, USA, 1999, p. 210–219.

Felizardo, K. R.; Nakagawa, E. Y.; Feitosa, D.; Minghim, R.; Maldonado, J. C. An approach based on visual text mining to support categorization and classification in the systematic mapping. In: *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering*, EASE'10, Swinton, UK, UK: British Computer Society, 2010, p. 34–43 (*EASE'10*,).

Felizardo, K. R.; Salleh, N.; Martins, R. M.; Mendes, E.; MacDonell, S. G.; Maldonado, J. C. Using visual text mining to support the study selection activity in systematic literature reviews. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011, p. 77–86.

Felizardo, K. R.; Souza, S. R. S.; Maldonado, J. C. The use of visual text mining to support the study selection activity in systematic literature reviews: A replication study. In: *3rd International Workshop on Replication in Empirical Software Engineering Research (RESER)*, 2013, p. 91–100.

Ferreira, L. P.; Vergilio, S. R. Tdsgen: An environment based on hybrid genetic algorithms for generation of test data. In: *17th International Conference on Software Engineering and Knowledge Engineering*, Taipei: Skokie, IL, USA : Knowledge Systems Institute Graduate School, 2005, p. 312–317.

Giacometti, C.; Souza, S. R. S.; Souza, P. S. L. Teste de mutação para a validação de aplicações concorrentes usando PVM. *Revista Eletrônica de Iniciação Científica - Sociedade Brasileira de Computação*, v. 02, n. 3, 2002.

Gill, A. *Introduction to the theory of finite-state machine.* New York: McGraw-Hill, 1962.

Gligoric, M.; Jagannath, V.; Marinov, D. Mutmut: Efficient exploration for mutation testing of multithreaded code. In: *Third International Conference on Software Testing, Verification and Validation (ICST)*, Paris, France, 2010, p. 55 –64.

Gligoric, M.; Zhang, L.; Pereira, C.; Pokam, G. Selective mutation testing for concurrent code. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, New York, NY, USA: ACM, 2013, p. 224–234 (*ISSTA 2013*,).
Disponível em `http://doi.acm.org/10.1145/2483760.2483773`

Harel, D.; Pinnel, A.; Schmidt, J. P.; Sherman, R. On the formal semantics of statecharts. In: *2nd IEEE Symposium on Logic in Computer Science*, Thaca, New York, 1987, p. 54–64.

Harrold, M. J. Testing: A roadmap. In: *22th International Conference on Sotware Engineering, in Future of Software Engineering (ICSE'2000)*, 2000, p. 61–72.

Hausen, A. C.; Vergilio, S. R.; Souza, S.; Souza, P.; Simao, A. A tool for structural testing of MPI programs. In: *8th IEEE Latin-American Test Workshop*, 2007, p. 1–5.

Hausen, A. C.; Vergílio, S. R.; Souza, S. R. S.; Souza, P. S. L.; Simao, A. S. Valimpi: Uma ferramenta para teste de programas paralelos. In: *Sessão de Ferramentas - XX SBES - Simpósio Brasileiro de Engenharia de Software*, Florianópolis, SC, (CDROM), 2006, p. 1–6.

Herman, P. M. A data flow analysis approach to program testing. *Australian Computer Journal*, v. 8, n. 3, 1976.

Hong, S.; Ahn, J.; Park, S.; Kim, M.; Harrold, M. J. Testing concurrent programs to achieve high synchronization coverage. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, New York, NY, USA: ACM, 2012, p. 210–220 (*ISSTA 2012*,).

Howden, W. E. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, v. 2, p. 208–215, 1976.

Hwang, G.-H.; Lin, H.-Y.; Lin, S.-Y.; Lin, C.-S. Statement-coverage testing for nondeterministic concurrent programs. In: *TASE*, 2012, p. 263–266.

Jagannath, V.; Gligoric, M.; Lauterburg, S.; Marinov, D.; Agha, G. Mutation operators for actor systems. In: *3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010)*, Paris, France, 2010, p. 157–162.

Kamil, A.; Yelick, K. Enforcing textual alignment of collectives using dynamic checks. *Lecture Notes in Computer Science (LNCS)*, v. 5898, p. 368–382, 2010.

Kitchenham, B. *Procedures for performing systematic reviews.* Relatório Técnico se-0401 (keele), Software Engineering Group - Department of Computer Science - Keele University and Empirical Software Engineering - National ICT Australia Ltd, Keele/Staffs-UK and Eversleigh-Australia, 2004.

Koppol, P. V.; Carver, R. H.; Tai, K.-C. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering*, v. 28, n. 6, p. 607–623, 2002.

Krawczyk, H.; Kuzora, P.; Neyman, M.; Proficz, J.; Wiszniewski, B. STEPS - a tool for testing PVM programs. In: *3rd SEIHPC Workshop*, citeseer.ist.psu.edu/357124.html, 1998.

Krena, B.; Letko, Z.; Vojnar, T.; Ur, S. A platform for search-based testing of concurrent software. In: *International Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2010)*, Trento, Italy, 2010, p. 48–58.

Kundu, S.; Ganai, M. K.; Wang, C. Contessa: Concurrency testing augmented with symbolic analysis. *Lecture Notes in Computer Science (LNCS)*, v. 6174, p. 127–131, 2010.

Křena, B.; Letko, Z.; Vojnar, T. Coverage metrics for saturation-based and search-based testing of concurrent software. In: *Proceedings of the Second International Conference on Runtime verification*, RV'11, Berlin, Heidelberg: Springer-Verlag, 2012, p. 177–192 (*RV'11*,).

Laski, J. W.; Korel, B. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, v. 9, n. 3, 1983.

Lei, Y.; Carver, R. H. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, v. 32, n. 6, p. 382–403, 2006.

Lemos, O. A. L.; Vincenzi, A. M. R.; Maldonado, J. C.; Masiero, P. C. Data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, v. 80, 2006.

Lu, S.; Jiang, W.; Zhou, Y. A study of interleaving coverage criteria. In: *Proceedings of the ACM SIGSOFT symposium on the foundations of software engineering*, New York, NY, USA: ACM, 2007, p. 533–536.

Maldonado, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software.* Tese de Doutoramento, DCA/FEE/UNICAMP, 1991.

Maldonado, J. C.; Delamaro, M. E.; Souza, S. R. S. Análise de mutantes: Uma avaliação empírica do axioma de antiextensionalidade. In: *Workshop de Qualidade de Sofware, co-alocado ao SBES - Simpósio Brasileiro de Engenharia de Software*, Recife, 1995, p. 136–140.

Maldonado, J. C.; Fabbri, S. C. P. F.; Mendonça, M.; Dória, E.; Martimiano, L. A. F.; Carver, J. Comparing code reading and testing criteria. In: *ISESE 2006 - International Symposium on Empirical Software Engineering*, Rio de Janeiro, 2006, p. 42–44.

Mathur, A. P. Performance, effectiveness, and reliability issues in software testing. In: *XV Annual International Computer Software and Applications Conference*, Tokio, Japão, 1991, p. 604–605.

Mathur, A. P.; Wong, W. E. Evaluation of the cost alternate mutation strategies. In: *VII Simpósio Brasileiro de Engenharia de Software (VII SBES)*, Rio de Janeiro, RJ, 1993.

Mathur, A. P.; Wong, W. E. An empirical comparison of data flow and mutation-based test adequacy criteria. *The Journal of Software Testing, Verification and Reliability*, v. 4, n. 1, p. 9–31, 1994.

McMinn, P. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, v. 14, n. 2, p. 105–156, 2004.

Melo, S. M.; Souza, S. R. S.; Souza, P. S. L. Structural testing for multithreaded programs: an experimental evaluation of the cost, strength and effectiveness. In: *24th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, San Francisco Bay, USA, 2012, p. 476–479.

Musuvathi, M.; Qadeer, S.; Ball, T.; Basler, G.; Nainar, P. A.; Neamtiu, I. Finding and reproducing heisenbugs in concurrent programs. In: *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, p. 267 – 280.

Myers, G. J. *The art of software testing.* 2 ed. New York: John Wiley & Sons, 2004.

Nguyen, C. D.; Perini, A.; Tonella, P.; Kessler, F. B. Constraint-based evolutionary testing of autonomous distributed systems. In: *IEEE International Conference on Software Testing Verification and Validation Workshop(ICSTW '08)*, Washington, DC, USA: IEEE Computer Society, 2008, p. 221–230.

Pallavi, J.; Mayur, N.; Chang-Seo, P.; Koushik, S. CalFuzzer: An extensible active testing framework for concurrent programs. In: Society, I. C., ed. *21st International*

Conference on Computer Aided Verification (CAV 2009), Grenoble, França, 2009, p. 675–681.

Petrenko, A.; Bochmann, G. V.; Yao, M. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems, Special Issue on Protocol Testing*, v. 29, n. 1, p. 81–106, 1996.

Pitarello, B.; Souza, S. Módulo de avaliação de seqüências de teste da ferramenta SCCTool, SIICUSP - Trabalho de Iniciação Científica - ICMC/ USP, São Carlos, 2006.

Prado, M. P.; Campanha, D. N.; Souza, S. R. S.; Maldonado, J. C. Um conjunto de artefatos para apoio à definição de estudos experimentais em teste de software. In: *ESELAW'2008 - 5th Experimental Software Engineering Latin American Workshop*, Salvador, 2008, p. 1–10.

Prado, M. P.; Souza, S. R. S.; Maldonado, J. C. Resultados de um estudo de caracterização e avaliação de critérios de teste estruturais entre os paradigmas procedimental e OO. In: *ESELAW'2010 - VII Experimental Software Engineering Latin American Workshop*, Goiania, 2010, p. 90–99.

Pressman, R. S. *Software engineering - a practitioner's approach.* sixth edition ed. McGraw-Hill, 2005.

Pugh, W.; Ayewah, N. Unit testing concurrent software. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, p. 513–516.

Rapps, S.; Weyuker, E. J. Selecting software test data using data flow information. *IEEE Transaction Software Engineering*, v. 11, n. 4, p. 367–375, 1985.

Remenska, D.; Templon, J.; Willemse, T.; Homburg, P.; Verstoep, K.; Casajus, A.; Bal, H. From UML to process algebra and back: An automated approach to model-checking software design artifacts of concurrent systems. In: *NASA Formal Methods*, v. 7871 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 244–260, 2013.

Rodrigues, E. M.; Oliveira, F. M.; Costa, L. T.; Bernardino, M.; Souza, S. R. S.; Saad, R.; Zorzo, A. F. Model-based testing applied to performance testing: An empirical study. *Empirical Software Engineering*, v. ACEITO, 2014.

Rungta, N.; Mercer, E. G. Clash of the titans: tools and techniques for hunting bugs in concurrent programs. In: *Proc. of PADTAD '09*, New York, NY, USA: ACM, 2009a, p. 9:1–9:10.

Rungta, N.; Mercer, E. G. A meta heuristic for effectively detecting concurrency errors. *Lecture Notes in Computer Science (LNCS)*, v. 5394, p. 23–37, 2009b.

Rungta, N.; Mercer, E. G.; Visser, W. Efficient testing of concurrent programs with abstraction-guided symbolic execution. *Lecture Notes in Computer Science (LNCS)*, v. 5578, p. 174–191, 2009.

Santiago, V.; Vijaykumar, N.; Ferreira, E.; Guimaraes, D.; Costa, R. C. Gtsc: Automated model-based test case generation from statecharts and finite state machines. In: *Sessão de Ferramentas do III Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*, 2012, p. 25–30.

Santiago, V.; Vijaykumar, N.; Guimaraes, D.; Amaral, A.; Ferreira, E. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. In: *IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008, p. 63–72.

Sarmanho, F. S.; Souza, P. S. L.; Souza, S. R. S.; Simao, A. S. Aplicação de teste estrutural para programas multithreads baseados em semáforos. In: *LPTD2007 - Workshop on Languages and Tools for Parallel and Distributed Programming - 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2007)*, Gramado, RS, cDROM, 2007, p. 18–21.

Sarmanho, F. S.; Souza, P. S. L.; Souza, S. R. S.; Simao, A. S. Structural testing for semaphore-based multithread programs. In: *ICCS2008 - International Conference on Computational Science, Lecture Notes in Computer Science*, Krakow: Springer-Verlag, 2008, p. 337–346.

Sen, A.; Abadir, M. S. Coverage metrics for verification of concurrent systemc designs using mutation testing. In: *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Anaheim, CA, United States, 2010, p. 75 –81.

Sherman, E.; Dwyer, M. B.; Elbaum, S. Saturation-based testing of concurrent programs. In: *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, New York, United States: ACM, 2009, p. 53–62.

Silva, R. A.; Souza, S. R. S.; Souza, P. S. L. Execução determinística de programas concorrentes durante o teste de mutação. In: *VI SAST - Workshop Brasileiro de Teste de Software Sistemático e Automatizado*, Natal, Brazil, 2012a, p. 1 –10.

Silva, R. A.; Souza, S. R. S.; Souza, P. S. L. Mutation operators for concurrent programs MPI. In: *13th Latin American Test Workshop*, Quito, Ecuador, 2012b, p. 69–74.

Silva, R. A.; Souza, S. R. S.; Souza, P. S. L. Utilizando uma arquitetura de referência para apoiar o desenvolvimento de uma ferramenta de teste de programas concorrentes. In: *CONTECSI - 10th International Conference on Information Systems and Technology Management*, Sao Paulo, Brazil, 2013, p. 1 –8.

Silva-Barradas, S. *Mutation analysis of concurrent software.* Tese de Doutoramento, Dottorato di Ricerca in Ingegneria Informatica e Automatica, Politecnico di Milano, 1998.

Simao, A.; Petrenko, A. Checking completeness of tests for finite state machines. *IEEE Transactions on Computers*, v. 593, 2010.

Simao, A.; Petrenko, A.; Maldonado, J. C. Comparing finite state machine test coverage criteria. *IET Software*, v. 3, p. 91–105, 2009.

Simao, A.; Souza, S. R. S.; Maldonado, J. C. A family of coverage testing criteria for coloured petri nets. In: *XVII Simpósio Brasileiro de Engenharia de Software*, Manaus, Brazil, 2003, p. 209–224.

Simao, A. S.; Maldonado, J. C. Mutation based test sequence generation for petri nets. In: *Workshop of Formal Methods (IVX SBES)*, João Pessoa, 2000.

Simao, A. S.; Vincenzi, A. M. R.; Maldonado, J. C.; Santana, A. C. L. Idel: A language for program instrumentation. In: *Conferencia Latinoamericana de Informática CLEI*, Montevideo, 2002, p. 1–12.

Souza, P. S. L.; Sawabe, E. T.; Simao, A. S.; Vergilio, S. R.; Souza, S. R. S.; Sarmanho, F. ValiPVM - a graphical tool for structural testing of PVM programs. In: *15th Euro PVM/MPI - LNCS - Recents Advances in Parallel Virtual Machine and Message Passing Interface*, Dublin, Ireland: Springer Berlin/Heidelberg, 2008a, p. 257–264.

Souza, P. S. L.; Souza, S. R. S.; Rocha, M. G.; Prado, R. R.; Batista, R. N. Data flow testing in concurrent programs with message-passing and shared-memory paradigms. In: *ICCS - International Conference on Computational Science*, Barcelona, Espanha, 2013, p. 149–158.

Souza, P. S. L.; Souza, S. R. S.; Zaluska, E. Structural testing for message-passing concurrent programs: an-extended test model. *Concurrency and Computation*, v. 26, n. 1, p. 21–50, 2014a.

Souza, S. R. S.; Brito, M. A. S.; Silva, R. A.; Souza, P. S. L.; Zaluska, E. Research in concurrent software testing: A systematic review. In: *PADTAD - Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging - in conjunction*

with ISSTA - International Symposium on Software Testing and Analysis, Toronto, Canada, 2011a, p. 1–5.

Souza, S. R. S.; Fabbri, S. C. P. F.; Barbosa, E. F.; Chaim, M. L.; Vincenzi, A. M. R.; Delamaro, M. E.; Jino, M.; Maldonado, J. C. *Introdução ao teste de software*, v. 1, cap. Estudos Teóricos e Experimentais M. E. Delamaro and J. C. Maldonado and M. Jino, p. 251–268, 2007a.

Souza, S. R. S.; Maldonado, J. C.; Fabbri, S. C. P. F. FCCE: Uma família de critérios de teste para validação de sistemas especificados em estelle. In: *Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro, 2001, p. 256–271.

Souza, S. R. S.; Maldonado, J. C.; Fabbri, S. C. P. F.; Masiero, P. C. Statecharts specifications: A family of coverage testing criteria. In: *Conferência Latino Americana de Informática (CLEI2000)*, Cidade de México, México, 2000a, p. 1–12.

Souza, S. R. S.; Maldonado, J. C.; Fabbri, S. C. P. F.; de Souza, W. L. Mutation testing applied to estelle specifications. *Software Quality Journal*, v. 8, n. 4, p. 285–302, kluwer Academic Publishers, 2000b.

Souza, S. R. S.; Prado, M. P.; Barbosa, E. F.; Maldonado, J. C. An experimental study to evaluate the impact of the programming paradigm in the testing activity. *CLEI Electronic Journal*, v. 15, n. 3, p. 1–13, 2012.

Souza, S. R. S.; Souza, P. S. L.; Brito, M. A. S.; Simao, A. S.; Zaluska, E. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. *Software Testing, Verification and Reliability*, v. ACEITO, 2014b.

Souza, S. R. S.; Souza, P. S. L.; Machado, M. C. C.; Camillo, M. S.; Simao, A. S.; Zaluska, E. Using coverage and reachability testing to improve concurrent program testing quality. In: *23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, Eden Roc Renaissance Miami Beach, United States, 2011b, p. 207–212.

Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L. *Introdução ao teste de software*, v. 1, cap. Teste de Programas Concorrentes M. E. Delamaro and J. C. Maldonado and M. Jino, p. 231–248, 2007b.

Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L.; Simao, A. S.; Bliscosque, T. G.; Lima, A. M.; Hausen, A. C. Valipar: A testing tool for message-passing parallel programs. In: *International Conference on Software knowledge and Software Engineering (SEKE05)*, Taipei-Taiwan, 2005, p. 386–391.

Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L.; Simao, A. S.; Hausen, A. C. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience*, v. 20, p. 1893–1916, 2008b.

Sugeta, T.; Maldonado, J.; Wong, E. W. Mutation testing applied to validate sdl specifications. In: *16th IFIP International Conference on Testing of Communicating Systems (TestCom2004)*, Oxford, United Kingdom: Lecture Notes on Computer Science n. 2978, 2004, p. 193–208.

Sun, T.; Ye, X.; Liu, J. A test generation method based on model reduction for parallel software. In: *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, 2012, p. 777–782.

Takahashi, J.; Kojima, H.; Furukawa, Z. Coverage based testing for concurrent software. In: *Distributed Computing Systems Workshops, 2008. ICDCS '08. 28th International Conference on*, 2008, p. 533–538.

Tasiran, S.; Keremoğlu, M. E.; Muşlu, K. Location pairs: a test coverage metric for shared-memory concurrent programs. *Empirical Software Engineering*, v. 17, n. 3, p. 129–165, 2012.

Taylor, R. N.; Levine, D. L.; Kelly, C. Structural testing of concurrent programs. *IEEE Transaction Software Engineering*, v. 18, n. 3, p. 206–215, 1992.

Turner, K. J. *Using formal description techniques – an introduction to Estelle, Lotos and SDL.* John Wiley & Sons, 1993.

Ural, H. Formal methods for test sequence generation. *Computer Communications*, v. 15, n. 5, p. 311–325, 1992.

Ural, H.; Saleh, K.; Willians, A. Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications*, v. 23, n. 7, 2000.

Ural, H.; Yang, B. Structural test selection criterion. *Information Processing Letters*, v. 28, p. 157–163, 1988.

Valgrind-Developers Valgrind-3.6.1. [*On-line*], http://valgrind.org, accessed: jun/2011.

Vergilio, S. R.; Souza, S. R. S.; Souza, P. S. L. Coverage testing criteria for message-passing parallel programs. In: *LATW2005 - 6th IEEE Latin-American Test Workshop*, Salvador, Ba, 2005, p. 161–166.

Vincenzi, A.; Delamaro, M. E.; Maldonado, J.; Wong, W. Establishing structural testing criteria for java bytecode. *Software Practice and Experience*, v. 36, n. 14, p. 1513–1541, 2006.

Weyuker, E. J. Axiomatizing software test data adequacy. *IEEE Trans. Softw. Eng.*, v. 12, n. 12, p. 1128–1138, 1986.

Weyuker, E. J. Testing component-based software: A cautionary tale. *IEEE Software*, v. 15, n. 5, 1998.

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B.; Wesslén, A. *Experimentation in software engineering: An introduction.* Kluwer Academic Publishers, 2000.

Wong, W. E.; Lei, Y.; Ma, X. Effective generation of test sequences for structural testing of concurrent programs. In: *10th IEEE International Conference on Engineering of Complex Systems (ICECCS 2005)*, Shanghai, China, 2005, p. 539–548.

Wong, W. E.; Maldonado, J. C.; Mathur, A. P. Mutation versus all-uses: An empirical evaluation of cost, strength, and effectiveness. In: *Software Quality and Productivity - Theory, Practice, Education and Training*, Hong Kong, 1994, p. 258–265.

Yang, C.-S.; Souter, A. L.; Pollock, L. L. All-du-path coverage for parallel programs. In: *International Symposium on Software Testing and Analysis (ISSTA'98), ACM-Software Engineering Notes*, 1998, p. 153–162.

Yang, C.-S. D.; Pollock, L. All-uses testing of shared memory parallel programs. *Software Testing, Verification and Reliability (STVR)*, v. 13, n. 1, p. 3–24, 2003.

Yang, C.-S. D.; Pollock, L. L. The challenges in automated testing of multithreaded programs. In: *14th International Conference on Testing Computer Software*, 1997, p. 157–166.

Yang, R.-D.; Chung, C.-G. Path analysis testing of concurrent programs. *Information and Software Technology*, v. 34, n. 1, 1992.

Yang, Y.; Chen, X.; Gopalakrishnan, G. *Inspect: A runtime model checker for multithreaded C programs.* Relatório Técnico, 2008.

Yu, J.; Narayanasamy, S. A case for an interleaving constrained shared-memory multi-processor. In: *36th Annual International Symposium on Computer Architecture (ISCA 2009)*, New York, United States: ACM, 2009, p. 325–336.

Zambianco, R. A.; Souza, S. R. S.; Souza, P. S. L. Reach-tree: Uma ferramenta para geração de Árvore de alcançabilidade para sistemas especificados em estelle. *Revista Eletrônica de Iniciação Científica - Sociedade Brasileira de Computação*, v. 02, n. 3, 2002.

Zhao, J. Data-flow-based unit testing of aspect-oriented programs. In: *XXVII IEEE Annual International Computer Software and Applications Conference - COMPSAC'2003*, Dallas - USA, 2003, p. 188–197.

Ziviani, N. *Projeto de algoritmos com implementações em java e c++*. Thomson, 2005a.

Ziviani, N. *Projeto de algoritmos com implementações em pascal e c*. 2nd. ed. Thomson, 2005b.

Apêndice

A

Lista com Publicações

Neste apêndice são apresentadas as principais publicações da autora, numeradas de acordo com o gráfico apresentado na Figura 2.1 do Capítulo 2. As publicações mais relevantes estão destacadas na Tabela A.1, assim como o nome da autora em cada referência.

Tabela A.1: Publicações relacionadas às pesquisas desenvolvidas pela autora após o término do seu doutorado. As publicações encontram-se numeradas de acordo com o gráfico apresentado nas Figuras 2.1 e 2.2. As publicações mais relevantes estão destacadas.

Num.	Referências
1	(Souza et al., 2014) Souza, P. S. L.; **Souza, S. R. S.**; Zaluska, E. Structural testing for message-passing concurrent programs: an extended test model. *Concurrency and Computation*, 26(1), p. 21- 50, 2014.
2	(Brito et al., 2012) Brito, M. A. S.; Rossi, J.; Braga, R. T. V.; **Souza, S. R. S.** An experience on applying software testing for teaching introductory programming courses. *CLEI Electronic Journal*, 15(4), p. 1-12, 2012.
3	(Souza et al. 2012) **Souza, S. R. S.**; Prado, M. P.; Barbosa, E. F.; Maldonado, J. C. An experimental study to evaluate the impact of the programming paradigm in the testing activity. *CLEI Electronic Journal*, 15(3), p. 1-13, 2012.
4	(Souza et al. 2008b) **Souza, S. R. S.**; Vergilio, S. R.; Souza, P. S. L.; Simao, A. S.; Hausen, A. C. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation*, 20, p. 1893-1916, 2008b.
5	(Zambiano et al. 2002) Zambianco, R. A.; **Souza, S. R. S.**; Souza, P. S. L. Reach-tree: Uma ferramenta para geração de árvore de alcançabilidade para sistemas especificados em estelle. *Revista Eletrônica de Iniciação Científica* - Sociedade Brasileira de Computação, 2(3), 2002.
6	(Giacometti et al. 2002) Giacometti, C.; **Souza, S. R. S.**; Souza, P. S. L. Teste de mutação para a validação de aplicações concorrentes usando PVM. *Revista Eletrônica de Iniciação Científica* - Sociedade Brasileira de Computação, 2(3), 2002.
7	(Brito et al. 2013) Brito, M. A. S.; **Souza, S. R. S.** An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs. In*: ICCS - International Conference on Computational Science*, Barcelona, Espanha ,2013, p. 250-259.
8	(Souza et al. 2013) Souza, P. S. L.; **Souza, S. R. S.**; Rocha, M. G.; Prado, R. R.; Batista, R. N. Data flow testing in concurrent programs with message-passing and shared-memory paradigms. In*: ICCS - International Conference on Computational Science*, Barcelona, Espanha, 2013, p. 149-158.
9	(Felizardo et al. 2013) Felizardo, K. R.; **Souza, S. R. S.**; Maldonado, J. C. The use of visual text mining to support the study selection activity in systematic literature reviews: A replication study. In: *3rd International Workshop on Replication in Empirical Software Engineering Research (RESER)*, 2013, p. 91-100.

Num.	Referências
10	(Silva et al. 2013) Silva, R. A.; **Souza, S. R. S.**; Souza, P. S. L. Utilizando uma arquitetura de referência para apoiar o desenvolvimento de uma ferramenta de teste de programas concorrentes. In*: 10th International Conference on Information Systems and Technology Management*, Sao Paulo, Brazil, 2013, p. 1-8.
11	(Silva et al. 2012a) Silva, R. A.; **Souza, S. R. S.**; Souza, P. S. L. Execução determinística de programas concorrentes durante o teste de mutação. In: *VI SAST - Workshop Brasileiro de Teste de Software Sistemático e Automatizado*, Natal, Brazil, 2012a, p. 1 -10.
12	(Silva et al. 2012b) Silva, R. A.; **Souza, S. R. S.**; Souza, P. S. L. mutation operators for concurrent programs in MPI. In: *13th Latin American Test Workshop*, Quito, Ecuador, 2012b, p. 69-74.
13	(Souza et al. 2011b) **Souza, S. R. S.**; Souza, P. S. L.; Machado, M. C. C.; Camillo, M. S.; Simao, A. S.; Zaluska, E. Using coverage and reachability testing to improve concurrent program testing quality. In: *23rd International Conference on Software Engineering and Knowledge Engineering*, Miami Beach, USA, 2011b, p. 207-212.
14	(Endo et al. 2010) Endo, A. T.; Lindshulte, M.; Simao, A. S.; **Souza, S. R. S.** Event and coverage-based testing of web services. In: *2nd Workshop on Model-Based Verification & Validation from Research to Practice* - in conjunction to the *4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, Cingapura, 2010, p. 1-8.
15	(Campanha et al. 2010) Campanha, D. N.; **Souza, S. R. S.**; Maldonado, J. C. Mutation testing in procedural and object-oriented paradigms: An evaluation of data structure programs. In: *XXIV Simpósio Brasileiro de Engenharia de Software, Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*, Salvador, 2010, p. 91-100.
16	(Cutigi et al. 2010) Cutigi, J. F.; Ribeiro, P. H.; Simao, A.; **Souza, S. R. S.** Redução do número de sequências no teste de conformidade de protocolos. In: *XI Workshop de Testes e Tolerância a Falhas*, Gramado (BR), 2010, p. 105-117.
17	(Prado et al. 2010) Prado, M. P.; **Souza, S. R. S.**; Maldonado, J. C. Resultados de um estudo de caracterização e avaliação de critérios de teste estruturais entre os paradigmas procedimental e OO. In: *ESELAW - VII Experimental Software Engineering Latin American Workshop*, Goiania, 2010, p. 90-99.
18	(Brito et al. 2010b) Brito, M. A. S.; Rossi, J.; **Souza, S. R. S.**; Braga, R. T. V. Reuso de conjuntos de casos de teste no ensino de disciplinas introdutórias de programação. In: *ESELAW - VII Experimental Software Engineering Latin American Workshop*, Goiania, 2010b, p. 101-110.

Num.	Referências
19	(Campanha et al. 2009) Campanha, D. N.; **Souza, S. R. S.**; Lemos, O. A. L.; Barbosa, E. F.; Maldonado, J. C. Reutilização de conjuntos de teste: Um estudo no domínio de algoritmos de ordenação. In: *ESELAW - VI Experimental Software Engineering Latin American Workshop*, São Carlos, 2009, p. 114 -123.
20	(Barbosa et al. 2008) Barbosa, E. F.; **Souza, S. R. S.**; Maldonado, J. C. An experience on applying learning mechanisms for teaching inspection and software testing. In: *21st Conference on Software Engineering Education & Training*, Charleston, South Carolina, 2008, p. 189-196.
21	(Sarmanho et al. 2008) Sarmanho, F. S.; Souza, P. S. L.; **Souza, S. R. S.**; Simao, A. S. Structural testing for semaphore-based multithread programs. In*: ICCS - International Conference on Computational Science*, Lecture Notes in Computer Science, 5101, Krakow: Springer-Verlag, 2008, p. 337-346.
22	(Prado et al. 2008) Prado, M. P.; Campanha, D. N.; **Souza, S. R. S.**; Maldonado, J. C. Um conjunto de artefatos para apoio à definição de estudos experimentais em teste de software. In: *ESELAW – 5th Experimental Software Engineering Latin American Workshop*, Salvador, 2008, p. 1-10.
23	(Souza et al. 2008a) Souza, P. S. L.; Sawabe, E. T.; Simao, A. S.; Vergilio, S. R.; **Souza, S. R. S.**; Sarmanho, F. ValiPVM - a graphical tool for structural testing of pvm programs. In: *15th EuroPVM/MPI*, Dublin, 2008a, p. 257-264.
24	(Endo et al. 2008) Endo, A. T.; Simao, A. S.; **Souza, S. R. S.**; Souza, P. S. L. Web services composition testing: A strategy based on structural testing of parallel programs. In: *TaicPart: Testing Academic & Industrial Conference - Practice and Research Techniques*, Windsor, 2008, p. 3-12.
25	(Hausen et al. 2007) Hausen, A. C.; Vergilio, S. R.; **Souza, S. R. S.** ; Souza, P. S. L.; Simao, A. A tool for structural testing of MPI programs. In: *8th IEEE Latin-American Test Workshop*, 2007, p. 1-5.
26	(Endo et al. 2007) Endo, A. T.; Simao, A. S.; **Souza, S. R. S.**; Souza, P. S. L. Aplicação de teste estrutural para composição de web services. In: *SAST - 1st Brazilian Workshop on Systematic and Automated Software Testing - XXI SBES - Simpósio Brasileiro de Engenharia de Software*, João Pessoa, PB, 2007, p. 13-20.
27	(Barbosa et al. 2007) Barbosa, E. F.; Domingues, **Souza, S. R. S.** ; Domingues, A. L. S.; Chan, A.; Nina, E.; Maldonado, J. C. Uma experiência no ensino de inspeção e teste de software. In: *VI Simpósio Brasileiro de Qualidade de Software (SBQS)*, Porto de Galinhas, Brazil, 2007, p. 309-324.

Num.	Referências
28	(Vergilio et al. 2005) Vergilio, S. R.; **Souza, S. R. S.**; Souza, P. S. L. Coverage testing criteria for message-passing parallel programs. In*: LATW - 6th IEEE Latin-American Test Workshop*, Salvador, Ba, 2005, p. 161-166.
29	(Souza et al. 2005) **Souza, S. R. S.**; Vergilio, S. R.; Souza, P. S. L.; Simao, A. S.; Bliscosque, T. G.; Lima, A. M.; Hausen, A. C. Valipar: A testing tool for message-passing parallel programs. In: *International Conference on Software knowledge and Software Engineering (SEKE05)*, Taipei-Taiwan, 2005, p. 386-391.
30	(Simao et al. 2003) Simao, A.; **Souza, S. R. S.**.; Maldonado, J. C. A family of coverage testing criteria for coloured petri nets. In: *XVII Simpósio Brasileiro de Engenharia de Software*, Manaus, Brazil, 2003, p. 209-224.
31	(Souza et al. 2001) **Souza, S. R. S.**; Maldonado, J. C.; Fabbri, S. C. P. F. FCCE: Uma família de critérios de teste para validação de sistemas especificados em Estelle. In: *Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro, 2001, p. 256-271.
32	(Brito et al. 2010) Brito, M. A. S.; **Souza, S. R. S.** Avaliação da efetividade dos critérios de teste estruturais no contexto de programas concorrentes. In: *15º Workshop de Teses e Dissertações em Engenharia de Software (WTES), 24º Simpósio Brasileiro de Engenharia de Software (SBES), Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*, 2010, p. 31-36.
33	(Brito et al. 2010a) Brito, M. A. S.; Felizardo, K.; Souza, P. S. L.; **Souza, S. R. S.** Concurrent software testing: A systematic review. In: *22nd IFIP International Conference on Testing Software and Systems*, Natal BR, 2010a, p. 79-84.
34	(Melo et al. 2012) Melo, S. M.; **Souza, S. R. S.**; Souza, P. S. L. Structural testing for multithreaded programs: an experimental evaluation of the cost, strength and effectiveness. In: 24th International Conference on Software Engineering and Knowledge Engineering (SEKE), San Francisco Bay, USA, 2012, p. 476-479.
35	(Souza et al. 2011a) **Souza, S. R. S.**; Brito, M. A. S.; Silva, R. A.; Souza, P. S. L.; Zaluska, E. Research in concurrent software testing: A systematic review. In: *PADTAD - Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* - in conjunction to the *ISSTA - International Symposium on Software Testing and Analysis*, Toronto, Canada, 2011a, p. 1-5.
36	(Sarmanho et al. 2007) Sarmanho, F.; Souza, P. S. L. ; **Souza, S. R. S.** ; Simao, A. Aplicação de teste estrutural para programas multithreads baseados em semáforos. In*: LPTD - Workshop on Languages and Tools for Parallel and Distributed Programming - 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Gramado, RS, 2007, p. 18-21.

Num.	Referências
37	(Hausen et al. 2006) Hausen, A. C.; Vergilio, S. R.; **Souza, S. R. S**; Souza, P. S. L. ; Simao, A. ValiMPI: Uma ferramenta para teste de programas paralelos. In: *Sessão de Ferramentas - XX SBES – Simpósio Brasileiro de Engenharia de Software*, Florianópolis, SC, 2006, p. 1-6.
38	(Souza et al. 2001) **Souza, S. R. S.**; Maldonado, J. C. Validação de Especificações de Sistemas Reativos: Definição e Análise de Critérios de Teste. In: *XIV CTD – Concurso de Teses e Dissertações – Congresso da Sociedade Brasileira de Computação*, 2001, p. 1-3.
39	(Souza et al. 2007a) **Souza, S. R. S.**; Fabbri, S. C. P. F.; Barbosa, E. F.; Chaim, M. L.; Vincenzi, A. M. R.; Delamaro, M. E.; Jino, M.; Maldonado, J. C. *Introdução ao teste de software*, v. 1, cap. Estudos Teóricos e Experimentais M. E. Delamaro and J. C. Maldonado and M. Jino, p. 251-268, 2007a.
40	(Souza et al. 2007b) **Souza, S. R. S.**; Vergilio, S. R.; Souza, P. S. L. *Introdução ao teste de software*, v. 1, cap. Teste de Programas Concorrentes M. E. Delamaro and J. C. Maldonado and M. Jino, p. 231-248, 2007b.

Apêndice

B

(Souza et al., 2014) Souza, P. S. L.; Souza, S. R. S.; Zaluska, E. Structural testing for message-passing concurrent programs: an extended test model. Concurrency and Computation, 26(1), p. 21-50, 2014.

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **26**:21–50
Published online 10 October 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2937

Structural testing for message-passing concurrent programs: an extended test model

Paulo S. L. Souza [1,*,†], Simone R. S. Souza [1] and Ed Zaluska [2]

[1] *Computer Systems Department, University of São Paulo, São Carlos, 13566-590, Brazil*
[2] *Electronic and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK*

SUMMARY

Developing high-quality, error-free message-passing concurrent programs is not trivial. Although a number of different primitives with associated semantics are available to assist such development, they often increase the complexity of the testing process. In this paper, we extend our previous test model for message-passing programs and present new structural testing criteria, taking into account additional features used in this paradigm, such as collective communication, non-blocking sends, distinct semantics for non-blocking receives, and persistent operations. Our new model also recognizes that sender primitives cannot always be matched with every receive primitive. This improvement allows us to remove statically a significant number of infeasible synchronization edges that would otherwise have to be analyzed later by the tester. In this paper, the test model is presented using the Message-Passing Interface standard; however, our new model has been designed to be flexible, and it can be configured to support a range of different message-passing environments or languages. We have carried out case studies showing the applicability of the new test model to represent message-passing programs and also to reveal errors, mainly those errors related to inter-process communication. In addition to increasing the number of features supported by the test model, we have also reduced the overall cost of testing significantly. Our case studies suggest that the number of synchronization edges can be reduced by up to 93%, mainly by eliminating infeasible edges between unmatchable communication primitives. The main contribution of the paper is to present a more flexible test model that provides improved coverage for message-passing programs and at the same time reduces the cost of testing significantly. Copyright © 2012 John Wiley & Sons, Ltd.

Received 1 August 2011; Revised 10 September 2012; Accepted 16 September 2012

KEY WORDS: concurrent software testing; message-passing programs; structural testing criteria

1. INTRODUCTION

Developing high-quality, error-free concurrent programs is not trivial. In addition to the usual sequential control and data flows found in sequential programs [1], concurrent algorithms have another added dimension because of the communication and synchronization requirements. The validation and testing of concurrent programs are a valuable mechanism to ensure quality by revealing unknown errors. However, features such as non-determinism, deadlocks, different programming paradigms, communication, and synchronization all make the testing activity for concurrent programs significantly more complex. For sequential programs, many testing problems were significantly reduced with the introduction of testing criteria and the implementation of supporting tools. A testing criterion is a predicate to be satisfied by a set of test cases that can be used as a guideline for the generation of test data. In other words, structural criteria allow the

*Correspondence to: Paulo S. L. Souza, USP/ICMC/SSC Avenida Trabalhador São-Carlense, 400, Centro, P.O. Box: 668, São Carlos, São Paolo, 13566-590, Brazil.
†E-mail: pssouza@icmc.usp.br

use of structural aspects from the source code to guide the selection of test cases. They are usually based on a control flow graph (CFG) and the definition or use of program variables [2].

Testing concurrent programs, in addition to the normal sequential testing, also requires exercising the different possible synchronizations with the same test input. When designing message-passing concurrent programs, developers must select which communication primitives are more appropriate for each specific use. Depending on the standard used, there may be different primitives and semantics available, which impacts on the performance and can increase the complexity of the development and testing activity. For instance, the Message-Passing Interface (MPI) standard [3, 4] has two different message-passing communication models. The one-sided communication model enables the sharing of the remote memory of another process and thus disassociates inter-process communication from the process synchronization mechanism. In other words, the behavior of the one-sided communication model is similar to the shared-memory paradigm often used in multithreaded programs. On the other hand, the two-sided communication model is based on a defined set of primitives responsible for both data exchange and process synchronization. Senders and receivers participate explicitly in the communication, using different types of point-to-point or collective primitives. Some other examples of primitives and semantics associated with the two-sided communication model are persistent, blocking and non-blocking, synchronous and asynchronous, buffered and un-buffered, and ready. These primitives and semantics are also affected by *communicators* that effectively determine the message reach.

Extending previous work on sequential program structural testing criteria, we have proposed structural testing criteria for the validation of concurrent programs, applicable to both message-passing [5] and shared-memory software [6]. These testing criteria are designed to exploit the available information about the control, data, and communication flows of concurrent programs, with both the sequential and parallel aspects taken into account.

These criteria improve significantly the quality of the test cases, providing a coverage measure that can be used in two important testing procedures. In the first one, the criteria can be used to guide the generation of test cases; in other words, the criteria are used as guidelines for test data selection. The second testing procedure is related to the evaluation of a test set; in this case, the criteria can be used to determine when the testing activity can be terminated on the basis of sufficient coverage of the required elements. The main contribution of the proposed testing criteria is to provide an efficient coverage measure for evaluating the testing activity progress and the quality of test cases.

The message-passing test model proposed in [5] represents accurately the standard blocking *send*/*receive* point-to-point primitives. *Non-blocking receives* can be represented as well because a primitive is active only until it returns from the call (after its return, another *receive* must be executed to recover a late message). This test model also considers that collective primitives can be accurately represented as a sequence of blocking point-to-point primitives, which execute the same expected semantic. However, message-passing programs are usually composed of distinct communication primitives that also need be tested, and our previous model was not able to represent these primitives adequately.

To overcome this problem, we have extended our previous test model for message-passing programs in different ways and now present new structural testing criteria. Some examples of these extensions are related to the primitives responsible for the following features: collective communication, non-blocking sends, distinct semantics for non-blocking receives, and persistent operations.

In this paper, the extended test model is presented in terms of the MPI two-sided communication model. MPI defines the syntax and the semantics of library routines to allow the development of portable message-passing programs using sequential languages, such as C and Fortran. MPI-2, the second version of the MPI standard, has become the *de facto* standard for writing parallel applications [3, 4], mainly because of its comprehensive features and large number of successful implementations [7–10].

Our new model anticipates that sends cannot always match with every receive and adjusts this representation to the reality found in typical message-passing programs. This improvement allows us to remove statically a significant amount of infeasible synchronization edges (sync-edges), which would otherwise be analyzed later by the tester. Our main contribution in this paper is firstly to

DOI: 10.1002/cpe

provide a more flexible test model that ensures a better coverage for message-passing programs and secondly at the same time significantly reduce the testing activity cost.

The remainder of this paper is organized as follows. In Section 2, we present basic concepts and also the previous test model, criteria, and ValiPar tool [5]. Section 3 describes the new structural test model, discussing the important aspects of the new features that have been included. We present our new testing criteria in Section 4 and the results when applied to two message-passing concurrent programs in Section 5. In Section 6, other works related to this paper are discussed. Our concluding remarks are presented in Section 7.

2. PREVIOUS WORK

2.1. Test model and basic concepts

The test model proposed in [5] captures control, data, and communication information. The model assumes that a fixed and known number of processes n are created at the initialization of the concurrent application. Each of these processes may execute different programs. However, they execute their own code in their own memory space. The concurrent program is defined by a set of n parallel processes $Prog = \{p^0, p^1, \ldots p^{n-1}\}$. Each process p has its own CFG^p, built using the same concepts as traditional sequential software [2]. In other words, a CFG of a process p is composed of a set of nodes N^p and a set of edges E^p. Each node n in the process p is represented by the notation n^p and corresponds to a set of commands that are sequentially executed or can be associated to a communication primitive (send or receive). N_s and N_r are two subsets of N, composed of nodes associated with *send* and *receive* primitives, respectively. A set R_i^p is associated with each $n_i^p \in N_s$, such that R_i^p contains the nodes that can receive a message sent by node n_i^p. The model considers the representation of collective messages in terms of several point-to-point messages. Figure 1 illustrates this with a simplified example of how the model is able to convert an *MPI_Reduce* collective primitive into an equivalent code from point-to-point primitives. This example considers that all processes belonging to an *icomm communicator* run the reduce primitive and that process p^0 receives different *sb* values from the other processes and determines the minimum *sb* among them. The function *new_tag_for_this_reduce* in Figure 1(b) makes a new *tag* for each new *MPI_Reduce* executed, assuring that each *MPI_Reduce* equivalent code will only receive its own messages.

In this model, a non-blocking receive returns with or without the message, depending on whether the message has reached the process previously (i.e., before the execution of the primitive). If it did not arrive before the execution of the primitive, another receive primitive must be executed later to receive the message.

```
...
MPI_Reduce(&sb, &rb, 1, MPI_INT, \
           MPI_MIN, p0, icomm);
...
```

(a)

```
...
tag = new_tag_for_this_reduce();
if(I am root) {
    rb = sb;
    for(i=0; i<NPROC-1; i++) {
        MPI_Recv(&b, 1, MPI_INT,\
            MPI_ANY_SOURCE, tag,\
              icomm, &sts);
        if (b<rb)
            rb=b;
    }
} else
    MPI_Send(&sb, 1, MPI_INT,
          root,tag,icomm);
...
```

(b)

Figure 1. Example of how to convert a Message-Passing Interface (MPI) collective primitive into an equivalent code on the basis of point-to-point primitives. Figure (a) has a reduce collective primitive assuming that the p^0 process is the root process, and figure (b) shows the equivalent code using send and receive basic primitives.

DOI: 10.1002/cpe

Prog is associated with a parallel CFG (PCFG), which is composed of the CFG^p (for $p = 0 \ldots n-1$) and a representation of the communication between the processes. A sync-edge $\left(n_i^p, n_j^q\right)$ links a *send* node in a process p to a *receive* node in a process q. The set of inter-process edges (E_s) is composed of sync-edges and represents the possibility of communication and synchronization between processes. (E) is the set containing all the edges, such that, $E = E_s \cup \bigcup_{p=0}^{n-1} E^p$.

An intra-process path π^p in a CFG^p has a finite sequence of nodes, $\pi^p = \left(n_1^p, n_2^p, \ldots n_m^p\right)$, where $\left(n_i^p, n_{i+1}^p\right) \in E^p$. An inter-process path $\Pi = \left(\pi^0, \pi^1, \ldots \pi^k, S\right)$ represents the concurrent execution of *Prog*, where S is the set of synchronization pairs executed, such that $S \subseteq E_s$. Synchronization pairs of S establish a conceptual path $\left(n_1^{p1}, n_2^{p1} \ldots n_i^{p1}, k_j^{p2} \ldots n_m^{p1}\right)$ or $\left(k_1^{p2}, k_2^{p2} \ldots n_i^{p1}, k_j^{p2} \ldots k_l^{p2}\right)$.

A variable x is defined when a value is stored into the correspondent memory position by the execution of a statement such as an assignment, input commands, and output parameters (when it is passed by reference to a function). The primitive receive performs a definition because it sets one (or more) variable(s) with a new value(s) received in the message. Set $def(n^p)$ is composed of the variables defined at n^p. A use of a variable occurs when the value associated with x is referenced. A *computational use* (*c-use*) occurs in a computation statement, a *predicate use* (*p-use*) occurs in a condition (predicate) associated with control flow statements related to an intra-process edge, and a *communication use* (*s-use*) occurs in a communication primitive related to an inter-process edge. A path $\pi = (n_1, n_2, \ldots, n_j, n_k)$ is definition clear with respect to (w.r.t.) a variable x from node n_1 to node n_k or edge (n_j, n_k), if $x \in def(n_1)$ and $x \notin def(n_i)$, for $i = 2 \ldots j$. Three kinds of associations establish pairs composed of definitions and uses of the same variables to be tested [2]: c-use association is defined by a triple (n^p, m^p, x), such that $x \in def(n^p)$, m^p has a c-use of x, and there is a definition-clear path w.r.t x from n^p to m^p; p-use association is defined by a triple (n^p, (m^p, k^p), x), such that $x \in def(n^p)$, (m^p, k^p) has a p-use of x, and there is a definition-clear path w.r.t x from n^p to (m^p, k^p); and s-use association is defined by a triple $(n^{p1}, (m^{p1}, k^{p2}), x)$, such that $x \in def(n^{p1})$, (m^{p1}, k^{p2}) has an s-use of x, and there is a definition-clear path w.r.t x from n^{p1} to (m^{p1}, k^{p2}).

p-use and c-use associations are intra-processes and are required when applying sequential testing criteria to each process separately. s-use are inter-process associations and allow the detection of communication faults (in the use of *send* and *receive* primitives). In this context, another inter-process association has been proposed to discover communication and synchronization faults: *s-c-use* association is given by $\left(n^{p1}, \left(m^{p1}, k^{p2}\right), l^{p2}, x^{p1}, x^{p2}\right)$, where there is an s-use association $\left(n^{p1}, \left(m^{p1}, k^{p2}\right), x^{p1}\right)$ and a c-use association $\left(k^{p2}, l^{p2}, x^{p2}\right)$; and *s-p-use* association is given by $\left(n^{p1}, \left(m^{p1}, k^{p2}\right), \left(n^{p2}, m^{p2}\right), x^{p1}, x^{p2}\right)$, where there is an s-use association $\left(n^{p1}, \left(m^{p1}, k^{p2}\right), x^{p1}\right)$ and a p-use association $\left(k^{p2}, \left(n^{p2}, m^{p2}\right), x^{p2}\right)$.

2.2. Structural testing criteria

A set of coverage testing criteria is defined from PCFG: *all-nodes*, *all-edges*, *all-nodes-r*, *all-nodes-s*, and *all-edges-s* (related to control and synchronization information) and *all-defs*, *all-defs-s*, *all-c-uses*, *all-p-uses*, *all-s-uses*, *all-s-c-uses*, and *all-s-p-uses* (related to data and communication information) [5]. The *all-nodes*, *all-edges*, *all-defs*, *all-c-uses*, and *all-p-uses* criteria were defined for sequential programs [2] and have been used by our test model [5]. The *all-nodes-r*, *all-nodes-s*, *all-edges-s*, *all-defs-s*, *all-s-uses*, *all-s-c-uses*, and *all-s-p-uses* are derived from sequential program testing criteria and were defined in [5]. In the context of this paper, the most significant criterion is the *all-edges-s*, which requires that the test set execute paths that cover all the sync-edge associations of the concurrent program under testing.

Required elements establish the minimal information that must be covered to satisfy a testing criterion. However, satisfying a testing criterion is not always possible because of the presence of infeasible elements. An element required by a criterion is infeasible if there is no set of values for the parameters, the input, and the global variables of the program that executes a path that covers that element. The determination of infeasible paths is an indeterminate problem [11].

DOI: 10.1002/cpe

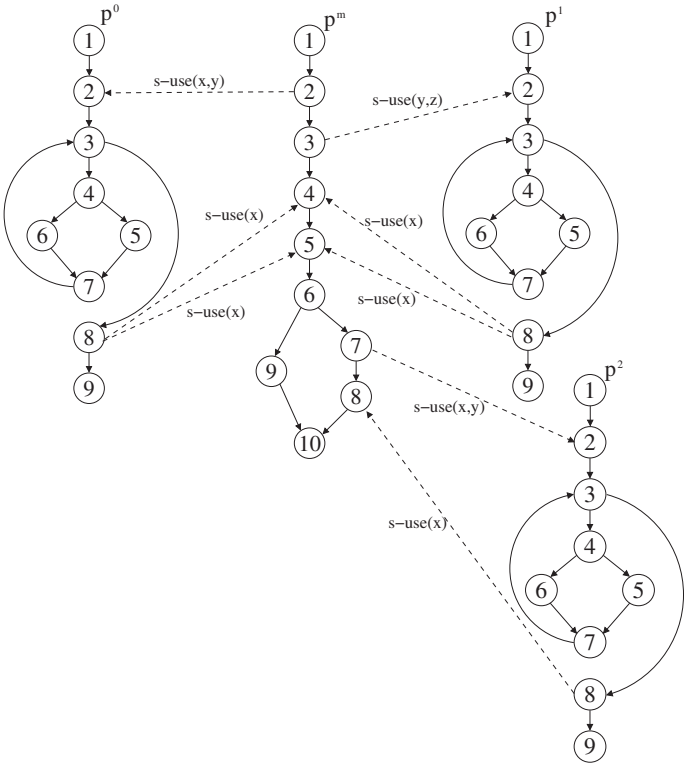


Figure 2. Example of a parallel control flow graph [5].

An example of a PCFG is shown in Figure 2. There are four processes, consisting of two different codes. Synchronization pairs are represented by dotted lines—for example, the pair $(2^0, 2^m)$ is one sync-edge between process p^0 and p^m. Each sync-edge is associated with one or more s-use associations and related to a variable represented in PCFG. Observe that, for reasons of simplicity, only some inter-process edges (and related s-use) are represented in Figure 2.

Non-determinism is the key issue addressed in this test model. As it is impossible to determine statically when a synchronization is feasible, a conservative approach is assumed, where every pair of send and receive events, which have the appropriate types, is considered as a possible matching pair. Thus, considering the example of Figure 2, the whole set of required sync-edges is $(2^m, 2^0)$, $(2^m, 2^1)$, $(2^m, 2^2)$, $(3^m, 2^0)$, $(3^m, 2^1)$, $(3^m, 2^2)$, $(7^m, 2^0)$, $(7^m, 2^1)$, $(7^m, 2^2)$, $(8^0, 4^m)$, $(8^0, 5^m)$, $(8^0, 8^m)$, $(8^0, 2^1)$, $(8^0, 2^2)$, $(8^1, 4^m)$, $(8^1, 5^m)$, $(8^1, 8^m)$, $(8^1, 2^0)$, $(8^1, 2^2)$, $(8^2, 4^m)$, $(8^2, 5^m)$, $(8^2, 8^m)$, $(8^2, 2^0)$, and $(8^2, 2^1)$. The sync-edges $(7^m, 2^0)$ and $(8^1, 8^m)$, for example, are infeasible but are required. With the use of the controlled execution feature [12] implemented in the ValiPar tool (described in the next section), it is possible to force the execution of feasible sync-edges. Notice that the non-determinism considered in this model is detected in a communication primitive, that is, sender or receiver nodes.

2.3. ValiPar testing tool

We have implemented ValiPar to support the effective application of the testing criteria [5, 13, 14]. We have adopted the concept of test sessions, which can be set up to test a given parallel program and allow for stopping testing activity and resuming it later. The tool allows the user to do the following: (i) create test sessions; (ii) save and execute test data; and (iii) evaluate the testing coverage w.r.t a given testing criterion. ValiPar is currently able to validate parallel programs written in Parallel Virtual Machine [15], MPI [16], Pthreads [6], and Business Process Execution Language (BPEL) [17].

DOI: 10.1002/cpe

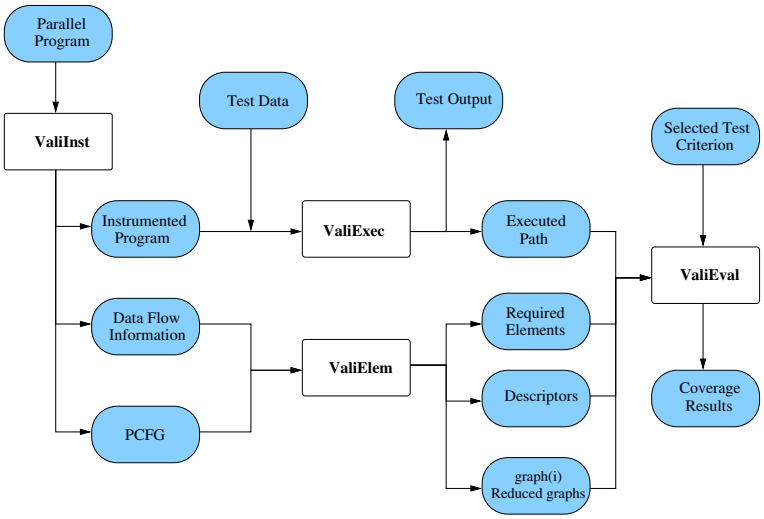


Figure 3. ValiPar tool architecture [5]. PCFG, parallel control flow graph.

The implementation of the tool follows the architecture shown in Figure 3. ValiPar has four main modules: *ValiInst* performs all static analysis of the parallel program, *ValiElem* generates the list of required elements, *ValiEval* performs the test case evaluation (coverage computation), and *ValiExec* considers the parallel program execution and generation of the executed paths.

The ValiInst module extracts control and data flow information and instruments the program. These tasks are supported by an instrumentation description language [18], a meta-language designed for program instrumentation. The PCFG is generated in text files, one for each function, with information about definitions and uses of variables in the nodes, as well as about occurrences of send and receive commands. The instrumented program is obtained by inserting checkpoint statements in the program being tested. These statements do not change the program semantics; they only write the necessary information into a trace file, recording the node and the process identifier of the send and receive commands. The instrumented program will produce the paths executed in each process, as well as the synchronization sequence produced within a test case.

The ValiElem module generates the elements required by the coverage testing criteria. These elements are generated from the PCFG using the information produced by ValiInst. Besides the PCFG, two other graphs are also used: (i) a reduced graph of heirs, to minimize the number of required edges; and (ii) a graph(i), to establish the associations of definitions and uses of variables, which are the required elements of the data-flow-based testing criteria.

The ValiExec module executes the instrumented program using the test data provided by the tester, which is responsible for the generation of the executable instrumented program. During the execution, the inputs and outputs of the program, command lines, execution traces, and synchronization sequences are stored in separate files. The execution trace includes the path executed in each process by the test input and is used during the evaluation of test cases to determine which required elements were covered. After the program execution, the tester can visualize the outputs and also the execution trace, to determine whether the output obtained is the same as expected. If it is not, an error is identified, and it must be corrected before continuing the testing activity.

ValiExec also enables the controlled execution of the parallel program under test. This feature is useful for replaying the test activity. Controlled execution guarantees that two executions of the parallel program with the same input will produce exactly the same paths and the same synchronization sequences. The implementation of a controlled execution is based on the work of Carver and Tai [12], adapted to message-passing programs. Synchronization sequences of each process are gathered at run time by the instrumented checkpoints of blocking and non-blocking *send* and *receive* primitives. The latter is also subject to non-determinism, so each request is associated with

DOI: 10.1002/cpe

the number of times that it has been evaluated. This information and other program inputs are used to achieve a deterministic execution and thus ensure that each test case replay is identical.

The ValiEval module evaluates the coverage achieved by test sets w.r.t. a criterion selected by the tester. ValiEval uses the elements generated by ValiElem and the paths executed by the test cases. The coverage score and the list of covered elements for the selected test criterion are provided as outputs.

2.4. The cost problem

The approach proposed in the previous work uses static analysis of the program under test to extract the relevant information for testing, which is then used to generate the appropriate information for coverage testing. However, a significant problem is the large number of infeasible elements generated that must still be analyzed, mainly because all possible interleaving between send–receive pairs are grouped in the same set of sync-edges. This set includes primitives related to both blocking/non-blocking point-to-point messages and collective primitives (because these are represented using several point-to-point messages).

Complete determination of infeasible elements is an extremely difficult problem, and it is not possible to determine such elements automatically. Souza *et al*. [19] described an approach to reduce this problem based on the work of Lei and Carver [20], which is essentially complementary to the approach described in the current paper. Lei and Carver did not address how to select the test case that will be used for the initial run, whereas we use the static analysis of the program to select in advance the optimum test cases. Lei and Carver also proposed a method based on reachability testing to obtain all of the executable synchronizations of a concurrent program from a given program execution, to reduce the number of redundant synchronizations. As this method uses dynamic information, only feasible synchronizations are generated, which is a considerable advantage. However, it also generates a large number of possible combinations of synchronization.

In [19], we propose a complementary approach, using reachability testing to target coverage testing for synchronization events. The essential concept is to make use of information about synchronizations provided by the coverage criterion to decide which race variants will be executed, selecting only those synchronizations that have not already been covered by existing test cases. Therefore, it is possible to execute each synchronization at least once and to use reachability testing to select only those synchronizations that are actually possible.

The new test model described in the next section has the advantage of reducing the number of infeasible sync-edges generated and thus improves the overall efficiency of the testing process.

3. EXTENDING THE TEST MODEL AND CONCEPTS

The test model proposed in [5, 13, 14] and improved in [6, 17, 19] represents a significant advance to guide the structural testing activity through distributed applications. It is robust and flexible enough to represent programs written in MPI, Process Virtual Machine, PThreads, and BPEL, as long as they use the basic primitives of communication and synchronization. However, some features were not considered in our original model, which makes the testing process harder or even prevents the correct representation of some primitives. These features are the use of different semantics for non-blocking receives, non-blocking sends, persistent communication primitives, collective communication primitives, inter-communicators, nodes having the behavior of both sender and receiver in the same execution, and sends that are not able to reach all the receives and considering for testing not just the communication primitives (such as send and receive) but all the primitives specified by the MPI-2 standard [3, 4].

We extended our testing model by considering this scenario. The sets used to represent testing elements have been modified, and new criteria have been proposed. Our aims are to improve the flexibility and robustness of the model and to accommodate the variety of semantics observed in typical MPI message-passing programs.

We define the set N_{std} as a subset of N representing nodes having primitives belonging to the MPI standard. The previous model considered only point-to-point communication primitives,

DOI: 10.1002/cpe

making testing harder when considering the concurrent source code as a whole. This N_{std} subset is useful not just for those criteria requiring to cover MPI primitives but, in particular, also for considering derived data types, packing/unpacking data through primitives other than send/receive, and dynamic generation of processes during the execution. Although these primitives are often encountered in MPI programs, they were not considered in our previous model. In respect to the dynamic generation of processes, our extended structural test model can be applied in concurrent processes that spawn new instances of MPI applications, through primitives such as *MPI_Comm_spawn*. However, the number of processes to be created must have been determined previously to the testing execution because the model is based on a static approach. To overcome this limitation, other tests can be performed considering distinct amounts of concurrent processes.

The extended model considers that the communication among processes uses two basic sets of primitives. The first is the set of *point-to-point communication* primitives, where a process sends a message using basic primitives such as *send* and *receive*. The second is named *collective communication*, where the message passing considers a group of processes. Communication through these two sets of primitives occurs in a predefined domain (or context), including all processes or just a subset of them.

Point-to-point communication primitives are still represented by the notations `send(p,k,t)` and `receive(p,k,t)`, as before. In these primitives, the process p sends (or receives) a message with tag t to (or from) the process k. Now, collective communication primitives can be represented with a sequence of several basic *sends*/*receives* or by specific primitives with notations, such as `Bcast(p,c)` and `Gather(p,c)`. In these two examples in particular, the process p uses *Bcast* to send the same message to a group of processes belonging to a context c, and in an analogous way, it uses *Gather* to catch data from a group of processes belonging to the same context c, sending these data to the process p. Observe that all processes in context c run the collective primitives and that they present different semantics, sometimes as senders and other times as receivers. This semantic depends on the parameter p, which can be dynamically instantiated during execution.

Figure 4 illustrates the collective communications expected for the new model: (a) one to all, (b) all to one, (c) all to all, and (d–f) reduction operation. In these examples, the node 8^0 is considered the *root* of the interaction, when applicable. The reduction operation has its computation represented in a hexagonal node, which encapsulates the operation performed during the communication. Double arrows represent nodes sending and receiving data in the same execution. Distinct collective primitives can also be joined, creating a new primitive. The *all-reduce* is an example of a primitive created from *reduce* and *broadcast* semantics.

Our previous model considers that non-blocking receives attempt to receive a message and then return, finishing the operation, whether the message has arrived or not. However, the primitive *MPI_Irecv* initiates a non-blocking receive and then returns without completing the operation. This non-blocking receive allows the MPI system to start writing data into the receive buffer, which should not be accessed until the operation is completed. The primitives *MPI_Test* and *MPI_Wait* or any of the other derived functions (*MPI_*{*Test*|*Wait*}{*all*|*some*|*any*}) are used to verify the communication status (non-blocking operation) or wait for its completion (blocking operation). The semantics of this MPI feature mean that it is not necessary to run other receive primitives to catch the message. However, a programming fault could result in incorrect behavior, where a message was received by a receive node different than the expected one. This behavior is possible, for example, if the primitives checking the communication status or waiting for the receive completion are not executed properly. This scenario is considered in our extended model during the execution of the instrumented code. The primitives responsible for starting the non-blocking receive and for verifying completion are linked statically, by considering the control and data flows; the execution trace provides sufficient information about the coverage of this non-blocking receive.

N_{wait}, N_{get_status}, and N_{test} are new sets used to represent nodes having primitives to wait and verify the completion of non-blocking requests. N_{iprobe} represents nodes having primitives to check, in a non-blocking way, whether a message is available to be received or not. The sets N_{get_status}, N_{test}, and N_{iprobe} have been created to support new criteria associated with those primitives responsible for the non-blocking verification of the end of sent or received messages. The N_{wait} set helps to determine if a blocking primitive *MPI_Wait*{*all*|*some*|*any*} was executed after a non-blocking

DOI: 10.1002/cpe

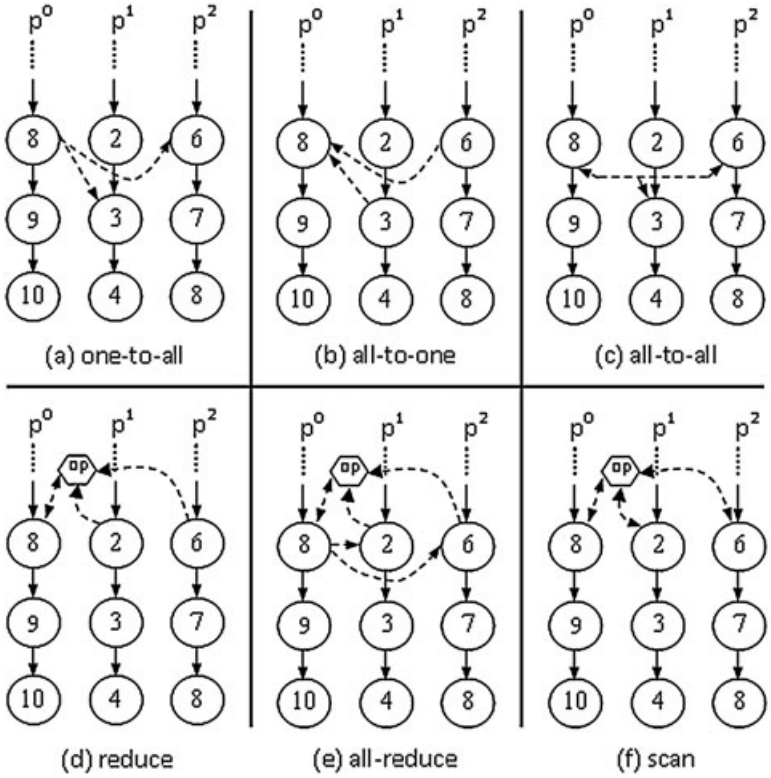


Figure 4. Examples of collective communication considered by our extended test model. In (a), node 8^0 sends messages to 3^1 and 6^2. In (b), node 8^0 receives messages from 3^1 and 6^2. In (c), nodes 8^0, 3^1, and 6^2 send and receive messages from each other. In (d), nodes 8^0, 2^1, and 6^2 send messages with data to the operation $< op >$, and the final result is forwarded to 8^0. In (e), nodes 8^0, 2^1, and 6^2 send messages with data to the operation $< op >$, and the final result is forwarded to the three processes. In (f), nodes 8^0, 2^1, and 6^2 send messages with data to the operation $< op >$, and the partial results are sent to the processes according to their *rank*.

communication primitive. The completion of a non-blocking primitive is a non-deterministic event, and the concurrent processes can present different behaviors depending on whether the completion check returns true or false. Our previous model assumed non-determinism only for communication primitives, such as point-to-point sends and receives. Although non-determinism obviously occurs in the previous model (because of synchronization among processes), the extended model allows considerably more flexibility in its representation of non-determinism because now the primitives *MPI_Test*{*all*|*some*|*any*}, *MPI_Iprobe*, and *MPI_Request_get_status* are also affected by non-deterministic behavior.

Send primitives can also be non-blocking. Exactly as with MPI non-blocking receives, a non-blocking send only initiates the communication, associating it with a request handle, which is used later to verify completion. The same primitives for checking and waiting for a non-blocking receive can also be used for a non-blocking send. It is very important to verify the communication status by waiting for completion after a non-blocking send because this signals that the message buffer is available and can then be re-used safely. The return of a (blocking) send primitive, in our original test model, represents the end of the operation, and the sender process can access any part of the send buffer after the send call returns. This non-blocking send is now considered in our extended test model, using the same methodology proposed for non-blocking receivers. The non-blocking send primitives are related to the primitives responsible for verifying completion in a static way, considering the control and data flows; the execution trace determines which non-blocking sends have been covered successfully.

DOI: 10.1002/cpe

It is important to emphasize that there are eight different sends in MPI, for point-to-point communications. The send has four different modes: standard, buffered, synchronous, and ready; and each one can be either blocking or non-blocking. These four send modes were considered in our original test model, which encapsulates the requirements for the following features: buffer, synchronization, and a previous execution of a receive after a send has been executed.

The MPI optimizes communications by binding the list of communication parameters into a persistent communication (with non-blocking semantics), requesting it once and then repeatedly using the request to initiate and complete the message. The primitive *MPI_Send_init*, *MPI_Bsend_init*, *MPI_Ssend_init*, or *MPI_Rsend_init* creates a persistent send request, and *MPI_Recv_init* creates a persistent receive request, binding to them all the arguments of the operation. The primitive *MPI_Start* or *MPI_Startall* initiates the non-blocking communication with a persistent request handle, and the primitive *MPI_Test*$\{all|some|any\}$ or *MPI_Wait*$\{all|some|any\}$ verifies whether the communication has been completed or not. Messages sent through a persistent send can be received by a non-persistent receive and *vice versa*.

Our original test model did not consider persistent communication at all. The new sets $N_{\text{recv\_init}}$ and $N_{\text{send\_init}}$ have now been inserted into our new model, representing nodes having primitives responsible for initializing persistent sends and receives. Set N_{start} has nodes with primitives to begin the persistent requests already configured. These sets are used to match the primitives associated with persistent communications in the code.

The previous model considers that a node is either a sender or a receiver, but not both. However, a primitive is allowed to be both a sender and a receiver in the MPI standard. The *Sendrecv*(p,k,t,k',p,t') primitive, similar to the *MPI_Sendrecv* [1, 3, 4], is an example of a primitive with these semantics where the process p sends a blocking message to k using the label t and, at the same execution, receiving other message from process k' with label t'. Collective primitives also have this behavior in MPI. They are able to set themselves as senders, receivers, or senders/receivers during execution, through their input arguments. In this case, techniques of data flow analysis could then distinguish if a primitive is a sender or a receiver [21, 22].

For this potential functionality to be accommodated, in our extended model, each primitive can be a sender, receiver, or sender/receiver. Each one can still be blocking or non-blocking. This is implemented by setting $T = \{$*'BS'*, *'NS'*, *'BR'*, *'NR'*, *'BSR'*, *'NSR'*$\}$ to define the semantics for *blocking sender*, *non-blocking sender*, *blocking receiver*, *non-blocking receiver*, *blocking sender–receiver*, and *non-blocking sender–receiver*, respectively. Although the MPI standard (version 2.2) does not define a non-blocking sender–receiver primitive, our extended test model nevertheless makes provision for this possibility, to improve the applicability to a wider range of different message-passing environments or languages.

Our previous test model matches all receive nodes for each send node. This conservative approach generates a large amount of infeasible sync-edges and, consequently, infeasible s-uses, s-c-uses, and s-p-uses. However, the MPI standard does not allow the matching of every communication primitives with every other primitive. Examples of this limitation are that point-to-point with collective primitives is not permitted; also, different collective primitives cannot be used arbitrarily among themselves. The generation of these infeasible sync-edges, resulting from communication primitives that are not able to interact with each other, can be reduced by clustering those primitives in subsets. Our model considers that all point-to-point primitives are labeled as 0 (indicating that they are able to interact), collective primitive *Bcast* are labeled as 1 (one), *Gather* as 2 (two), and so on. Alternatively, all the primitives can be inserted into just one set, allowing them to interact freely. These subsets provide additional flexibility for the model because they can encapsulate differences between different message-passing standards or languages. They also reduce the overall cost of applying the structural testing activity because the primitives can now be grouped, with their intra-communicators and inter-communicators taken into account. Set F gives the support for these clusters of primitives (or functions) in our extended model, where $F = \{f_0, f_1, \ldots f_{m-1}\}$ and m indicates the amount of different clusters.

Because of the restrictions introduced for the matching of different primitives and the possibility of send–receive nodes, the subsets N_{s} and N_{r} are no longer used to represent *send* and *receive* primitives, respectively. They have been replaced by a new set, N_{sync}, which avoids replicating the

DOI: 10.1002/cpe

representation of send-and-receive nodes and, at the same time, allows senders and receivers to be clustered in a straightforward fashion. The nodes in N_{sync} are represented by the triple $\left(n_i^p, f_g, t_u\right)$, where node $n_i^p \in N^p$ and has a communication primitive, $f_g \in F$, and $t_u \in T$.

To simplify our notation, we define a node n_i^p as a *sender node* if $n_i^p \in N_{\text{sync}} | \exists \left(n_i^p, f_g, t_u\right)$ with $t_u = (\text{'}BS\text{'} \vee \text{'}NS\text{'} \vee \text{'}BSR\text{'} \vee \text{'}NSR\text{'})$. Analogously, a blocking sender and a non-blocking sender are sender nodes in which $t_u = (\text{'}BS\text{'} \vee \text{'}BSR\text{'})$ and $t_u = (\text{'}NS\text{'} \vee \text{'}NSR\text{'})$, respectively. In the same way, the node is defined as a *receiver node* if $n_i^p \in N_{\text{sync}} \mid \exists \left(n_i^p, f_g, t_u\right)$ with $t_u = (\text{'}BR\text{'} \vee \text{'}NR\text{'} \vee \text{'}BSR\text{'} \vee \text{'}NSR\text{'})$. Analogously, a blocking receiver and a non-blocking receiver are receiver nodes in which $t_u = (\text{'}BR\text{'} \vee \text{'}BSR\text{'})$ and $t_u = (\text{'}NR\text{'} \vee \text{'}NSR\text{'})$, respectively.

Given these new features, a set R_i^p is associated with each $(n_i^p, f_g, t_u) \in N_{\text{sync}}$, such that

$$R_i^p = \left\{ n_j^k \in N_{\text{sync}} | n_i^p \text{ is a sender node and } n_j^k \text{ is a receiver node } \forall k \neq p \wedge k = 0 \ldots n-1 \right\}$$

that is, R_i^p contains the nodes that can receive a message sent by node n_i^p.

From our previous model and using the aforementioned definitions, we also define the following sets:

- The *set of inter-process edges* (E_{s}) contains edges that represent the communication between two processes, such that

$$E_{\text{s}} = \left\{ \left(n_j^{p1}, n_k^{p2}\right) \mid \left(n_j^{p1}, f_g, t_u\right) \in N_{\text{sync}} \wedge n_j^{p1} \text{ is a sender node} \wedge n_k^{p2} \in R_j^{p1} \right\}$$

- The *set of edges* (E) contains all edges, such that

$$E = E_{\text{s}} \cup \bigcup_{p=0}^{n-1} E^p$$

The intra-process path π^p, the inter-process path Π, and the set $def(n^p)$ with the definitions of variables have not been changed in our new test model. Similarly, the concepts of a definition-clear path and the use of variables remain unchanged. These concepts apply to the following associations, as defined previously: c-use, p-use, s-use, s-c-use, and s-p-use. s-use associations also consider collective primitives, and in this case, they can occur in a parallel way. In [5, 13, 14], more details about the definitions of the previous test model are found.

Given the new features now provided in our test model, we propose two more intra-process associations related to non-blocking communication primitives:

nr-use association: This association considers the uses of a variable x after it has been defined by a non-blocking receive. This association is given by (n^p, m^p, x, S), such that $x \in def(n^p)$, n^p is a non-blocking receiver node, m^p has a (c, p, or s)-use of x, and S is the set with all definition-clear paths w.r.t. x from n^p to m^p.

ns-use association: This association considers the uses or the definitions of a variable x after it has been used by a non-blocking send. This association is given by (n^p, m^p, x, S), such that there is an s-use of x in n^p, n^p is a non-blocking sender node (or $x \in def(m^p)$ or there is a (c, p, or s)-use of x in m^p), and S is the set with all definition-clear paths w.r.t. x from n^p to m^p.

These two new associations are used by two new testing criteria described in the next section. However, they are also helpful in identifying paths that do not contain a proper check on whether the non-blocking primitive has been completed or not. In this way, these associations provide a basis on which to implement warnings about the existence of paths without any test (red alert) and/or paths with only non-blocking tests (yellow alert) between a non-blocking primitive (sender/receiver) and the use/definition of the variable considered in that message.

4. NEW STRUCTURAL TESTING CRITERIA

The structural testing criteria for message-passing parallel programs proposed in our previous work have not been changed, apart from *all-nodes-s* and *all-nodes-r*, which are both now defined in

DOI: 10.1002/cpe

relation to N_{sync}. We also propose four new criteria, based on the control, data, and communication flows. These criteria allow the testing of new aspects in the parallel programs, which are now considered in our model.

The testing criteria based on the control and communication flows are as follows:

***all-nodes-s* criterion**: The test sets must execute paths that cover all the nodes $n_i^p \in N_{\text{sync}} \wedge n_i^p$ is a sender node.

***all-nodes-r* criterion**: The test sets must execute paths that cover all the nodes $n_i^p \in N_{\text{sync}} \wedge n_i^p$ is a receiver node.

***all-nodes-std* criterion**: The test sets must execute paths that cover all the nodes $n_i^p \in N_{\text{std}}$. The *all-nodes* criterion subsumes this new criterion; however, this more restrictive alternative is computationally less expensive and also useful in guiding the testing activity through the MPI primitives.

***all-nodes-nt* criterion**: The test sets must execute paths that cover every node $n_i^p \in N_{\text{test}} \vee n_i^p \in N_{\text{get\_status}} \vee n_i^p \in N_{\text{iprobe}}$ at least twice. The first execution must assume that the message reached the process after the testing node execution, whereas the second one must assume that the message reached it before. Controlled execution mechanisms must be employed here to force the required synchronization of these primitives.

The testing criteria based on data and message-passing flows are described in the following. These two new criteria require associations between a non-blocking primitive and the definition and/or uses of variables. The objective is to validate the data flow between the processes when a non-blocking primitive is used.

***all-nr-uses* criterion**: The test set must execute paths that cover all the *nr-use* associations.

***all-ns-uses* criterion**: The test set must execute paths that cover all the *ns-use* associations.

5. APPLYING THE NEW TEST MODEL AND CRITERIA

We will illustrate the modifications introduced in our test model with the assistance of a demonstration program (the ring program) written in C/MPI (Figures 5 and 6). The ring program is a multiple-program, multiple-data program composed of four concurrent processes (p^{m}, p^0, p^1, and p^2). It is designed to demonstrate, in a short program, the use of several different MPI features, such as dynamic processes creation, point-to-point/collective primitives, locking/non-blocking communication, persistent communication requests, different MPI send modes, intra-group/inter-group communication, and nodes acting as senders/receivers. The *master process* (p^{m}) creates dynamically three slaves (p^0, p^1, and p^2) and exchanges point-to-point messages with p^0, which starts a 'token ring' including the other two slaves. Each slave then generates a new token, and depending on the incoming argument, they determine by means of a reduction operation the minimum or maximum new value evaluated.

The node of the CFG for both the master and slave programs has been identified in the source code with a comment of the form '*/ *n* */' (where *n* is the CFG node). We identify an instance of any node with a superscript identifying either the master process (m) or a slave process (0, 1, or 2).

Node 5^{m} is responsible for creating the three slave processes (Figure 5), which run the *app slave* program (Figure 6). Process p^{m} then waits for the slaves in an *MPI_Barrier* (node 6^{m}). Slaves commence execution and at nodes 6^0, 6^1, and 6^2 run the *MPI_Barrier* for synchronization, by using an inter-communicator *icomm*. Process p^{m} sends the *token* to p^0 through a point-to-point *non-blocking send*, starting this communication at node 9^{m} and finishing it at node 10^{m}. Process p^0 receives this message by starting the non-blocking receive at node 10^0 and finishing it at node 11^0 or node 13^0, depending on whether the message will be available before node 11^0 is executed. This behavior is non-deterministic at node 11^0. The token ring is started and finished by p^0 (nodes 19^0 and 22^0), whereas this token is passed by p^1 and p^2 using nodes 24 up to 26. Observe that p^0 uses a persistent synchronous non-blocking send at node 19^0, which has been instantiated at node 8^0. Processes p^1 and p^2 use the same persistent send primitive at node 25^0.

DOI: 10.1002/cpe

```
/*****************************/
/* RING PROGRAM - MASTER CODE */
/*****************************/
#include<stdio.h>
#include<stdlib.h>
#include"mpi.h"
#define NUM_SLAVES 3
#define SLAVE_0 0
#define MASTER 0
int main(int argc, char **argv) {
       // argv[1] is 1 for minimum and 0 for maximum reducing
/*1*/  int my_rank,nproc,tkn,gtkn=-2,errcod[NUM_SLAVES],items;
/*1*/  int arrived,min;
/*1*/  MPI_Status sts;
/*1*/  MPI_Request reqid0, reqid1;
/*1*/  MPI_Comm icomm;
/*2*/  MPI_Init(&argc, &argv);
/*3*/  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
/*4*/  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
/*5*/  MPI_Comm_spawn("app_slave", MPI_ARGV_NULL, \
                      NUM_SLAVES,MPI_INFO_NULL, MASTER, \
                      MPI_COMM_WORLD,&icomm, errcod);\
/*6*/  MPI_Barrier(icomm);
/*7*/  tkn = atoi(argv[1]);
/*8*/  MPI_Recv_init(&min, 1, MPI_INT, SLAVE_0, 3, \
                     icomm, &reqid1);
/*9*/  MPI_Isend(&tkn, 1, MPI_INT, SLAVE_0, 1, \
                 icomm, &reqid0);
       /* doing something else */
/*10*/ MPI_Wait(&reqid0, &sts);
/*11*/ MPI_Get_count(&sts, MPI_INT, &items);
/*12*/ MPI_Start(&reqid1); // starting a non-blocking MPI_Recv_init for min
       /* doing something else */
/*13*/ MPI_Test(&reqid1, &arrived, &sts);
/*14*/ if (!arrived)    {
          /* still doing something else */
/*15*/    MPI_Wait(&reqid1, &sts);
/*16*/    MPI_Get_count(&sts, MPI_INT, &items);
/*17*/    arrived = -1;
       }
/*18*/ MPI_Isend(&min, 1, MPI_INT, SLAVE_0, 3, \
                 icomm, &reqid0);
       /* doing something else */
/*19*/ MPI_Wait(&reqid0, &sts);
/*20*/ MPI_Get_count(&sts, MPI_INT, &items);
/*21*/ if (min) {
/*22*/    MPI_Reduce(&tkn, &gtkn, items, MPI_INT, MPI_MIN, \
                     MPI_ROOT, icomm);
/*23*/    printf("Minimum=%d\n", gtkn);
       }
       else {
/*24*/    MPI_Reduce(&tkn, &gtkn, items, MPI_INT, MPI_MAX, \
                     MPI_ROOT, icomm);
/*25*/    printf("Maximum=%d\n", gtkn);
       }
/*26*/ MPI_Finalize();
/*27*/ exit(0);          }
```

Figure 5. Master process source code for the ring program.

When the ring is completed, p^0 sends back the value of *min* to the master process and receives it back. This rendezvous operation is carried out at node 23^0 using an *MPI_Sendrecv_replace* primitive. Process p^{m} initiates a non-blocking receive for this message at node 12^{m} (a persistent non-blocking receive primitive, instantiated at node 8^{m}). Again, this receive is completed at node 13^{m} or 15^{m}, and the behavior is non-deterministic.

Process p^{m} sends the token back to p^0 at node 18^{m} (finishing this communication at node 19^0). The message is received by p^0 at node 23^0, which then broadcasts it to the other slaves at node 28^0. Each slave evaluates a new token considering their *my_rank*. After that, all processes, including p^{m}, execute the *MPI_Reduce* primitive, and depending on the value of the *min* flag, they find the minimum or maximum new token just generated by the slaves. Process p^{m} receives the result of this operation at node 22^{m} or 24^{m} and then prints the final result. The new tokens evaluated by the slaves are $\{0, 1, 2\}$ when executing with the test input $\{1\}$, and the expected output is 'minimum = 0'. If the test input is $\{0\}$, the new tokens evaluated by slaves are $\{0, -1, -2\}$, and the expected output is 'maximum = 0'.

DOI: 10.1002/cpe

```
/****************************/
/* RING PROGRAM - SLAVE CODE */
/****************************/
#include<stdio.h>
#include<stdlib.h>
#include"mpi.h"
#define MASTER 0
#define SLAVE_0 0
int main(int argc, char **argv) {
/*1*/  int my_rank, nproc, next, tkn, gtkn, arrived, min,items;
/*1*/  MPI_Status sts;
/*1*/  MPI_Comm icomm;
/*1*/  MPI_Request reqid0, reqid1;
/*2*/  MPI_Init(&argc, &argv);
/*3*/  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
/*4*/  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
/*5*/  MPI_Comm_get_parent(&icomm);
/*6*/  MPI_Barrier(icomm);
/*7*/  next = (my_rank + 1) % nproc;
/*7*/  tkn = -1;
/*8*/  MPI_Ssend_init(&tkn, 1, MPI_INT, next, 2, MPI_COMM_WORLD, \
                      &reqid1);
/*9*/  if ( my_rank == 0 ) {
/*10*/     MPI_Irecv(&tkn,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,\
                     icomm,&reqid0);
           /* doing something else */
/*11*/     MPI_Test(&reqid0, &arrived, &sts);
/*12*/     if (!arrived)          {
              /* still doing something else */
/*13*/        MPI_Wait(&reqid0, &sts);
/*14*/        MPI_Get_count(&sts, MPI_INT, &items);
/*15*/        arrived=-1;
           }
/*16*/     tkn = tkn * tkn;
/*16*/     min = tkn;
/*17*/     if(!tkn)
/*18*/        tkn--;
/*19*/     MPI_Start(&reqid1); // starting a non-blocking MPI_Issend for tkn
           /* doing something else */
/*20*/     MPI_Wait(&reqid1, &sts);
/*21*/     MPI_Get_count(&sts, MPI_INT, &items);
/*22*/     MPI_Recv(&tkn, items, MPI_INT, MPI_ANY_SOURCE,\
                   MPI_ANY_TAG, MPI_COMM_WORLD,&sts);
/*23*/     MPI_Sendrecv_replace(&min,items,MPI_INT,MASTER,3, \
                     MPI_ANY_SOURCE,MPI_ANY_TAG,icomm,&sts);
         }
         else   {
/*24*/     MPI_Recv(&tkn, 1, MPI_INT, MPI_ANY_SOURCE, \
                    MPI_ANY_TAG, MPI_COMM_WORLD, &sts);
/*25*/     MPI_Start(&reqid1); // starting a non-blocking MPI_Issend for tkn
           /* doing something else */
/*26*/     MPI_Wait(&reqid1, &sts);
         }
/*27*/ MPI_Get_count(&sts, MPI_INT, &items);
/*28*/ MPI_Bcast(&min, items, MPI_UNSIGNED, SLAVE_0, MPI_COMM_WORLD);
/*29*/ tkn *= my_rank;
/*30*/ if (min) {
/*31*/     MPI_Reduce(&tkn, &gtkn, items, MPI_INT, MPI_MIN, \
                     MASTER, icomm);
         }
         else {
/*32*/     MPI_Reduce(&tkn, &gtkn, items, MPI_INT, MPI_MAX, \
                     MASTER, icomm);
         }
/*33*/ MPI_Finalize();
/*34*/ exit(0);            }
```

Figure 6. Slave process source code for the ring program.

The PCFG for the ring program is presented in Figure 7. Inter-process edges are represented by dotted lines. For simplicity, this figure contains only some of the inter-process edges (and related s-use). Table I presents the sets $def(n_i^p)$. Table II contains the values of all sets proposed in our test model.

In Table III, we present some elements required by the structural testing criteria proposed. Test inputs must be generated to exercise each possible required element. For example, considering the test input {1}, the execution path is $\Pi = (\pi^m, \pi^0, \pi^1, \pi^2, S)$, where $\pi^m = \{1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, 10^m, 11^m, 12^m, 13^m, 14^m, 15^m, 16^m, 17^m, 18^m, 19^m, 20^m, 21^m, 22^m, 23^m, 26^m, 27^m\}$ or $\pi^m = \{1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, 10^m, 11^m, 12^m, 13^m, 14^m, 18^m, 19^m, 20^m, 21^m, 22^m,$

DOI: 10.1002/cpe

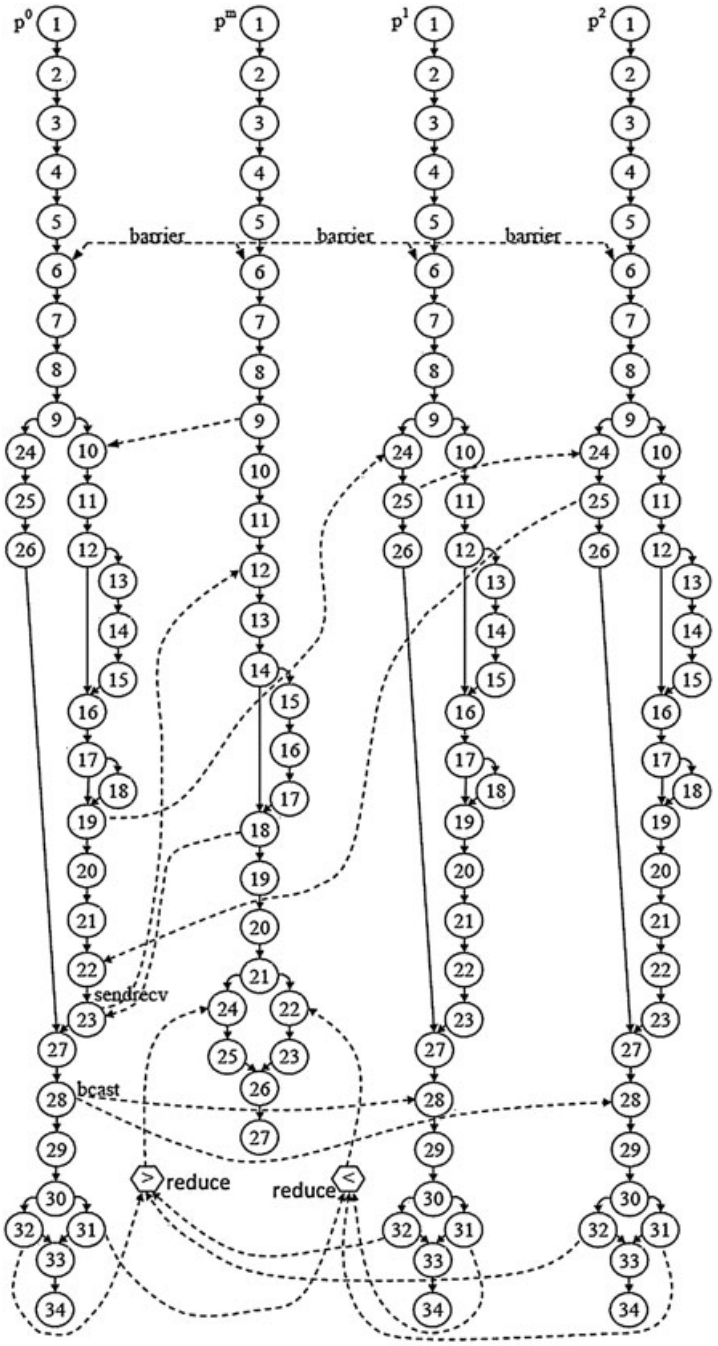


Figure 7. Parallel control flow graph for the ring program.

23^{m}, 26^{m}, $27^{m}\}$ depending on the non-deterministic test performed at node 13^{m}; $\pi^0 = \{1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 11^0, 12^0, 13^0, 14^0, 15^0, 16^0, 17^0, 19^0, 20^0, 21^0, 22^0, 23^0, 27^0, 28^0, 29^0, 30^0, 31^0, 33^0, 34^0\}$ or $\pi^0 = \{1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, 10^0, 11^0, 12^0, 16^0, 17^0, 19^0, 20^0, 21^0, 22^0, 23^0, 27^0, 28^0, 29^0, 30^0, 31^0, 33^0, 34^0\}$ depending on the non-deterministic test performed at node 12^0; $\pi^1 = \{1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 24^1, 25^1, 26^1, 27^1, 28^1, 29^1, 30^1, 31^1, 33^1, 34^1\}$; $\pi^2 = \{1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 24^2, 25^2, 26^2, 27^2, 28^2, 29^2, 30^2, 31^2, 33^2, 34^2\}$; and $S = \{(6^m, 6^0), (6^m, 6^1), (6^m, 6^2), (9^m, 10^0), (18^m, 23^0), (6^0, 6^m), (6^0, 6^1), (6^0, 6^2), (19^0, 24^1), (23^0, 12^m), (28^0, 28^1), (28^0, 28^2), (31^0, 22^m), (6^1, 6^m), (6^1, 6^0), (6^1, 6^2), (25^1, 24^2), (31^1, 22^m), (6^2, 6^m), (6^2, 6^0), (6^2, 6^1), (25^2, 22^0), (31^2, 22^m)\}$.

DOI: 10.1002/cpe

Because of the test input {1}, some required elements will not be exercised, such as the nodes 24^m, 25^m, 18^0, and 32^2 and the edges $(32^0, 24^m)$, $(32^1, 24^m)$, and $(32^2, 24^m)$. The test input {0} will be able to exercise these and other required elements. The non-determinism present in the ring program is not detected by the communication primitives, but by the non-blocking testing primitives at nodes 13^m, 11^0, 11^1, and 11^2. This occurs because the node starting the non-blocking receive is considered to be a receiver (node 10^0 in this case). The trace file registering the program execution is able to indicate if a test or wait primitive (node 11^0 or 13^0), associated with this non-blocking communication, confirms the sync-edge. This behavior is analogous for non-blocking sends. A controlled execution is applied in this case, exercising the required elements related to the non-determinism of the primitives checking the completion of non-blocking communications [5].

5.1. Revealing faults

The effectiveness (in terms of fault revealing) of the proposed criteria can be illustrated by examples of the faults that could be present in the ring program (Figures 5 and 6) and by demonstrating how the criteria contribute to revealing these faults. The faults are based on the works of Howden [23] and Krawczyk and Wisniewski [24], which describe typical faults in traditional and parallel programs, respectively.

Howden [23] introduced two types of faults in traditional programs: computation and domain faults. The first one occurs when the result of a computation for an input of the program domain is different from the expected result. The second one occurs when a path different from the expected one is executed. For example, in the slave source code, replacing the command of node 29^0 {*tkn* *= *my_rank*} by the incorrect command *tkn* *= (*my_rank* + 1) corresponds to a computation fault for both the possible test inputs. For the test input {0}, the incorrect output 'maximum = −1' is printed, and for the input {1}, the output 'minimum = 1' is printed. A domain fault can be illustrated by changing the statement if(*min*) at node 21^m of the slave source code by the incorrect predicate if(!*min*), with a different path taken during the execution. In this last case, the fault will be revealed, printing the wrong outputs 'minimum = −2' and 'maximum = 2' for the inputs {0} and {1}, respectively. These faults can be revealed by applying the traditional criteria (*all-edges*, *all-nodes*, etc.) and testing the CFG separately. These situations illustrate the importance of investigating the application of criteria for sequential testing in parallel software.

In the context of parallel programs, a computation fault can be related to a communication fault. To illustrate this effect, we consider the *MPI_Reduce* primitive at node 24^m (Figure 5). This reduction primitive initiates an *inter-communicator*, and then the root process belonging to the first group combines data from the processes in the second one to perform the requested operation. In our case, the master process must use *MPI_ROOT*, and the other ones (the slave processes in the second group) must use the *MASTER* argument to specify the rank of that root process in the first group. Now, consider replacing (erroneously) the argument *MPI_ROOT* by *MASTER* at node 24^m. This will prevent the master process from receiving the correct value from the reduction primitive. Both test cases {0} and {1} will reveal this fault, producing the value −2 as output. The uncovered s-use associations $(29^0, (31^0, 22^m), tkn, gtkn)$, $(29^1, (31^1, 22^m), tkn, gtkn)$, and $(29^2, (31^2, 22^m), tkn, gtkn)$, which should be covered, can guide the tester to reveal the mistake in relation to this primitive. The *all-edges-s* and *all-s-p-uses* criteria can also help the tester in this search.

Krawczyk and Wisniewski [24] presented two kinds of faults related to parallel programs: observability and locking faults. The observability fault is a special kind of domain fault, related to synchronization faults. These faults can be observed or not during the execution of the same test input; the observation depends on the synchronization of processes (i.e., non-determinism is present). Locking faults occur when the parallel program does not complete its execution, that is, it remains locked, waiting forever.

To illustrate the observability fault, we consider that node 13^0 (containing the *MPI_Wait* primitive) has been (erroneously) removed. In this case, when the message sent from node 9^m in the master process does not reach p^0 before the execution of the non-blocking testing at node 11^0, the non-blocking receive at node 10^0 fails, and the message can be received at node 22^0 instead (which does not receive the message from node 25^2 anymore). The *tkn* value remains with the

DOI: 10.1002/cpe

Table I. Definition sets for the ring program.

	Slaves		
Master	0	1	2
$def(1^{m}) = \{argc, argv, gtkn\}$	$def(1^{0}) = \{argc, argv\}$	$def(1^{1}) = \{argc, argv\}$	$def(1^{2}) = \{argc, argv\}$
$def(3^{m}) = \{my\_rank\}$	$def(3^{0}) = \{my\_rank\}$	$def(3^{1}) = \{my\_rank\}$	$def(3^{2}) = \{my\_rank\}$
$def(4^{m}) = \{nproc\}$	$def(4^{0}) = \{nproc\}$	$def(4^{1}) = \{nproc\}$	$def(4^{2}) = \{nproc\}$
$def(5^{m}) = \{icomm, errcod\}$	$def(5^{0}) = \{icomm\}$	$def(5^{1}) = \{icomm\}$	$def(5^{2}) = \{icomm\}$
$def(7^{m}) = \{tkn\}$	$def(7^{0}) = \{next, tkn\}$	$def(7^{1}) = \{next, tkn\}$	$def(7^{2}) = \{next, tkn\}$
$def(8^{m}) = \{reqid1\}$	$def(8^{0}) = \{reqid1\}$	$def(8^{1}) = \{reqid1\}$	$def(8^{2}) = \{reqid1\}$
$def(9^{m}) = \{reqid0\}$	$def(10^{0}) = \{tkn, reqid0\}$	$def(10^{1}) = \{tkn, reqid0\}$	$def(10^{2}) = \{tkn, reqid0\}$
$def(10^{m}) = \{reqid0, sts\}$	$def(11^{0}) = \{reqid0, arrived, sts\}$	$def(11^{1}) = \{reqid0, arrived, sts\}$	$def(11^{2}) = \{reqid0, arrived, sts\}$
$def(11^{m}) = \{items\}$	$def(13^{0}) = \{reqid0, sts\}$	$def(13^{1}) = \{reqid0, sts\}$	$def(13^{2}) = \{reqid0, sts\}$
$def(12^{m}) = \{min, requid1\}$	$def(14^{0}) = \{items\}$	$def(14^{1}) = \{items\}$	$def(14^{2}) = \{items\}$
$def(13^{m}) = \{reqid1, arrived, sts\}$	$def(15^{0}) = \{arrived\}$	$def(15^{1}) = \{arrived\}$	$def(15^{2}) = \{arrived\}$
$def(15^{m}) = \{reqid1, sts\}$	$def(16^{0}) = \{tkn, min\}$	$def(16^{1}) = \{tkn, min\}$	$def(16^{2}) = \{tkn, min\}$
$def(16^{m}) = \{items\}$	$def(18^{0}) = \{tkn\}$	$def(18^{1}) = \{tkn\}$	$def(18^{2}) = \{tkn\}$
$def(17^{m}) = \{arrived\}$	$def(19^{0}) = \{reqid1\}$	$def(19^{1}) = \{reqid1\}$	$def(19^{2}) = \{reqid1\}$
$def(18^{m}) = \{reqid0\}$	$def(20^{0}) = \{reqid1, sts\}$	$def(20^{1}) = \{reqid1, sts\}$	$def(20^{2}) = \{reqid1, sts\}$
$def(19^{m}) = \{reqid0, sts\}$	$def(21^{0}) = \{items\}$	$def(21^{1}) = \{items\}$	$def(21^{2}) = \{items\}$
$def(20^{m}) = \{items\}$	$def(22^{0}) = \{tkn, sts\}$	$def(22^{1}) = \{tkn, sts\}$	$def(22^{2}) = \{tkn, sts\}$
$def(22^{m}) = \{gtkn\}$	$def(23^{0}) = \{min, sts\}$	$def(23^{1}) = \{min, sts\}$	$def(23^{2}) = \{min, sts\}$
$def(24^{m}) = \{gtkn\}$	$def(24^{0}) = \{tkn, sts\}$	$def(24^{1}) = \{tkn, sts\}$	$def(24^{2}) = \{tkn, sts\}$
	$def(25^{0}) = \{reqid1\}$	$def(25^{1}) = \{reqid1\}$	$def(25^{2}) = \{reqid1\}$
	$def(26^{0}) = \{reqid1, sts\}$	$def(26^{1}) = \{reqid1, sts\}$	$def(26^{2}) = \{reqid1, sts\}$
	$def(27^{0}) = \{items\}$	$def(27^{1}) = \{items\}$	$def(27^{2}) = \{items\}$
	$def(28^{0}) = \{min\}$	$def(28^{1}) = \{min\}$	$def(28^{2}) = \{min\}$
	$def(29^{0}) = \{tkn\}$	$def(29^{1}) = \{tkn\}$	$def(29^{2}) = \{tkn\}$
	$def(31^{0}) = \{gtkn\}$	$def(31^{1}) = \{gtkn\}$	$def(31^{2}) = \{gtkn\}$
	$def(32^{0}) = \{gtkn\}$	$def(32^{1}) = \{gtkn\}$	$def(32^{2}) = \{gtkn\}$

DOI: 10.1002/cpe

Table II. Sets of the test model for the ring program with four processes (one master and three slaves).

Sets	Elements
$Prog$	$\{p^{m}, p^{0}, p^{1}, p^{2}\}$
N	$\{1^{m}, 2^{m}, 3^{m}, 4^{m}, 5^{m}, 6^{m}, 7^{m}, 8^{m}, 9^{m}, 10^{m}, 11^{m}, 12^{m}, 13^{m}, 14^{m}, 15^{m}, 16^{m}, 17^{m}, 18^{m}, 19^{m}, 20^{m}, 21^{m}, 22^{m}, 23^{m}, 24^{m}, 25^{m}, 26^{m}, 27^{m}, 1^{0}, 2^{0}, 3^{0}, 4^{0}, 5^{0}, 6^{0}, 7^{0}, 8^{0}, 9^{0}, 10^{0}, 11^{0}, 12^{0}, 13^{0}, 14^{0}, 15^{0}, 16^{0}, 17^{0}, 18^{0}, 19^{0}, 20^{0}, 21^{0}, 22^{0}, 23^{0}, 24^{0}, 25^{0}, 26^{0}, 27^{0}, 28^{0}, 29^{0}, 30^{0}, 31^{0}, 32^{0}, 33^{0}, 34^{0}, 1^{1}, 2^{1}, 3^{1}, 4^{1}, 5^{1}, 6^{1}, 7^{1}, 8^{1}, 9^{1}, 10^{1}, 11^{1}, 12^{1}, 13^{1}, 14^{1}, 15^{1}, 16^{1}, 17^{1}, 18^{1}, 19^{1}, 20^{1}, 21^{1}, 22^{1}, 23^{1}, 24^{1}, 25^{1}, 26^{1}, 27^{1}, 28^{1}, 29^{1}, 30^{1}, 31^{1}, 32^{1}, 33^{1}, 34^{1}, 1^{2}, 2^{2}, 3^{2}, 4^{2}, 5^{2}, 6^{2}, 7^{2}, 8^{2}, 9^{2}, 10^{2}, 11^{2}, 12^{2}, 13^{2}, 14^{2}, 15^{2}, 16^{2}, 17^{2}, 18^{2}, 19^{2}, 20^{2}, 21^{2}, 22^{2}, 23^{2}, 24^{2}, 25^{2}, 26^{2}, 27^{2}, 28^{2}, 29^{2}, 30^{2}, 31^{2}, 32^{2}, 33^{2}, 34^{2}\}$
N_{std}	$\{2^{m}, 3^{m}, 4^{m}, 5^{m}, 6^{m}, 8^{m}, 9^{m}, 10^{m}, 11^{m}, 12^{m}, 13^{m}, 15^{m}, 16^{m}, 17^{m}, 18^{m}, 19^{m}, 20^{m}, 22^{m}, 24^{m}, 26^{m}, 2^{0}, 3^{0}, 4^{0}, 5^{0}, 6^{0}, 8^{0}, 10^{0}, 11^{0}, 13^{0}, 14^{0}, 15^{0}, 19^{0}, 20^{0}, 21^{0}, 22^{0}, 23^{0}, 24^{0}, 25^{0}, 26^{0}, 27^{0}, 28^{0}, 31^{0}, 32^{0}, 33^{0}, 2^{1}, 3^{1}, 4^{1}, 5^{1}, 6^{1}, 8^{1}, 10^{1}, 11^{1}, 13^{1}, 14^{1}, 15^{1}, 19^{1}, 20^{1}, 21^{1}, 22^{1}, 23^{1}, 24^{1}, 25^{1}, 26^{1}, 27^{1}, 28^{1}, 31^{1}, 32^{1}, 33^{1}, 2^{2}, 3^{2}, 4^{2}, 5^{2}, 6^{2}, 8^{2}, 10^{2}, 11^{2}, 13^{2}, 14^{2}, 15^{2}, 19^{2}, 20^{2}, 21^{2}, 22^{2}, 23^{2}, 24^{2}, 25^{2}, 26^{2}, 27^{2}, 28^{2}, 31^{2}, 32^{2}, 33^{2}\}$
N_{sync}	$\{(6^{m}, 1, \textit{'BSR'}), (9^{m}, 0, \textit{'NS'}), (12^{m}, 0, \textit{'NR'}), (18^{m}, 0, \textit{'NS'}), (22^{m}, 2, \textit{'BSR'}), (24^{m}, 2, \textit{'BSR'}), (6^{0}, 1, \textit{'BSR'}), (10^{0}, 0, \textit{'NR'}), (19^{0}, 0, \textit{'NS'}), (22^{0}, 0, \textit{'BR'}), (23^{0}, 0, \textit{'BSR'}), (24^{0}, 0, \textit{'BR'}), (25^{0}, 0, \textit{'BS'}), (28^{0}, 3, \textit{'BSR'}), (31^{0}, 2, \textit{'BSR'}), (32^{0}, 2, \textit{'BSR'}), (6^{1}, 1, \textit{'BSR'}), (10^{1}, 0, \textit{'NR'}), (19^{1}, 0, \textit{'NS'}), (22^{1}, 0, \textit{'BR'}), (23^{1}, 0, \textit{'BSR'}), (24^{1}, 0, \textit{'BR'}), (25^{1}, 0, \textit{'BS'}), (28^{1}, 3, \textit{'BSR'}), (31^{1}, 2, \textit{'BSR'}), (32^{1}, 2, \textit{'BSR'}), (6^{2}, 1, \textit{'BSR'}), (10^{2}, 0, \textit{'NR'}), (19^{2}, 0, \textit{'NS'}), (22^{2}, 0, \textit{'BR'}), (23^{2}, 0, \textit{'BSR'}), (24^{2}, 0, \textit{'BR'}), (25^{2}, 0, \textit{'BS'}), (28^{2}, 3, \textit{'BSR'}), (31^{2}, 2, \textit{'BSR'}), (32^{2}, 2, \textit{'BSR'})\}$
N_{recv_init}	$\{8^{m}\}$
N_{send_init}	$\{8^{0}, 8^{1}, 8^{2}\}$
N_{start}	$\{12^{m}, 19^{0}, 25^{0}, 19^{1}, 25^{1}, 19^{2}, 25^{2}\}$
N_{wait}	$\{10^{m}, 15^{m}, 19^{m}, 13^{0}, 20^{0}, 26^{0}, 13^{1}, 20^{1}, 26^{1}, 13^{2}, 20^{2}, 26^{2}\}$
N_{test}	$\{13^{m}, 11^{0}, 11^{1}, 11^{2}\}$
N_{get_status}	$\{\emptyset\}$
N_{iprobe}	$\{\emptyset\}$
R_i^{m}	$R_6^{m} = \{6^{0}, 6^{1}, 6^{2}\}$, $R_9^{m} = \{10^{0}, 22^{0}, 23^{0}, 24^{0}, 10^{1}, 22^{1}, 23^{1}, 24^{1}, 10^{2}, 22^{2}, 23^{2}, 24^{2}\}$, $R_{18}^{m} = \{10^{0}, 22^{0}, 23^{0}, 24^{0}, 10^{1}, 22^{1}, 23^{1}, 24^{1}, 10^{2}, 22^{2}, 23^{2}, 24^{2}\}$, $R_{22}^{m} = \{31^{0}, 32^{0}, 31^{1}, 32^{1}, 31^{2}, 32^{2}\}$, $R_{24}^{m} = \{31^{0}, 32^{0}, 31^{1}, 32^{1}, 31^{2}, 32^{2}\}$
R_i^{0}	$R_6^{0} = \{6^{m}, 6^{1}, 6^{2}\}$, $R_{19}^{0} = \{12^{m}, 10^{1}, 22^{1}, 23^{1}, 24^{1}, 10^{2}, 22^{2}, 23^{2}, 24^{2}\}$, $R_{23}^{0} = \{12^{m}, 10^{1}, 22^{1}, 23^{1}, 24^{1}, 10^{2}, 22^{2}, 23^{2}, 24^{2}\}$, $R_{25}^{0} = \{12^{m}, 10^{1}, 22^{1}, 23^{1}, 24^{1}, 10^{2}, 22^{2}, 23^{2}, 24^{2}\}$, $R_{28}^{0} = \{28^{1}, 28^{2}\}$, $R_{31}^{0} = \{22^{m}, 24^{m}, 31^{1}, 32^{1}, 31^{2}, 32^{2}\}$, $R_{32}^{0} = \{22^{m}, 24^{m}, 31^{1}, 32^{1}, 31^{2}, 32^{2}\}$
R_i^{1}	$R_6^{1} = \{6^{m}, 6^{0}, 6^{2}\}$, $R_{19}^{1} = \{12^{m}, 10^{0}, 22^{0}, 23^{0}, 24^{0}, 10^{2}, 22^{2}, 23^{2}, 24^{2}\}$, $R_{23}^{1} = \{12^{m}, 10^{0}, 22^{0}, 23^{0}, 24^{0}, 10^{2}, 22^{2}, 23^{2}, 24^{2}\}$, $R_{25}^{1} = \{12^{m}, 10^{0}, 22^{0}, 23^{0}, 24^{0}, 10^{2}, 22^{2}, 23^{2}, 24^{2}\}$, $R_{28}^{1} = \{28^{0}, 28^{2}\}$, $R_{31}^{1} = \{22^{m}, 24^{m}, 31^{0}, 32^{0}, 31^{2}, 32^{2}\}$, $R_{32}^{1} = \{22^{m}, 24^{m}, 31^{0}, 32^{0}, 31^{2}, 32^{2}\}$
R_i^{2}	$R_6^{2} = \{6^{m}, 6^{0}, 6^{1}\}$, $R_{19}^{2} = \{12^{m}, 10^{0}, 22^{0}, 23^{0}, 24^{0}, 10^{1}, 22^{1}, 23^{1}, 24^{1}\}$, $R_{23}^{2} = \{12^{m}, 10^{0}, 22^{0}, 23^{0}, 24^{0}, 10^{1}, 22^{1}, 23^{1}, 24^{1}\}$, $R_{25}^{2} = \{12^{m}, 10^{0}, 22^{0}, 23^{0}, 24^{0}, 10^{1}, 22^{1}, 23^{1}, 24^{1}\}$, $R_{28}^{2} = \{28^{0}, 28^{1}\}$, $R_{31}^{2} = \{22^{m}, 24^{m}, 31^{0}, 32^{0}, 31^{1}, 32^{1}\}$, $R_{32}^{2} = \{22^{m}, 24^{m}, 31^{0}, 32^{0}, 31^{1}, 32^{1}\}$

DOI: 10.1002/cpe

Table II. *Continued.*

Sets	Elements
E_i^{m}	{(1^{m}, 2^{m}), (2^{m}, 3^{m}), (3^{m}, 4^{m}), (4^{m}, 5^{m}), (5^{m}, 6^{m}), (6^{m}, 7^{m}), (7^{m}, 8^{m}), (8^{m}, 9^{m}), (9^{m}, 10^{m}), (10^{m}, 11^{m}), (11^{m}, 12^{m}), (12^{m}, 13^{m}), (13^{m}, 14^{m}), (14^{m}, 15^{m}), (15^{m}, 16^{m}), (16^{m}, 17^{m}), (17^{m}, 18^{m}), (14^{m}, 18^{m}), (18^{m}, 19^{m}), (19^{m}, 20^{m}), (20^{m}, 21^{m}), (21^{m}, 22^{m}), (22^{m}, 23^{m}), (23^{m}, 26^{m}), (21^{m}, 24^{m}), (24^{m}, 25^{m}), (25^{m}, 26^{m}), (26^{m}, 27^{m})}
E_i^0	{(1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (5^0, 6^0), (6^0, 7^0), (7^0, 8^0), (8^0, 9^0), (9^0, 10^0), (10^0, 11^0), (11^0, 12^0), (12^0, 13^0), (13^0, 14^0), (14^0, 15^0), (15^0, 16^0), (12^0, 16^0), (16^0, 17^0), (17^0, 18^0), (18^0, 19^0), (17^0, 19^0), (19^0, 20^0), (20^0, 21^0), (21^0, 22^0), (22^0, 23^0), (23^0, 27^0), (9^0, 24^0), (24^0, 25^0), (25^0, 26^0), (26^0, 27^0), (27^0, 28^0), (28^0, 29^0), (29^0, 30^0), (30^0, 31^0), (31^0, 33^0), (30^0, 32^0), (32^0, 33^0), (33^0, 34^0) }
E_i^1	{(1^1, 2^1), (2^1, 3^1), (3^1, 4^1), (4^1, 5^1), (5^1, 6^1), (6^1, 7^1), (7^1, 8^1), (8^1, 9^1), (9^1, 10^1), (10^1, 11^1), (11^1, 12^1), (12^1, 13^1), (13^1, 14^1), (14^1, 15^1), (15^1, 16^1), (12^1, 16^1), (16^1, 17^1), (17^1, 18^1), (18^1, 19^1), (17^1, 19^1), (19^1, 20^1), (20^1, 21^1), (21^1, 22^1), (22^1, 23^1), (23^1, 27^1), (9^1, 24^1), (24^1, 25^1), (25^1, 26^1), (26^1, 27^1), (27^1, 28^1), (28^1, 29^1), (29^1, 30^1), (30^1, 31^1), (31^1, 33^1), (30^1, 32^1), (32^1, 33^1), (33^1, 34^1) }
E_i^2	{(1^2, 2^2), (2^2, 3^2), (3^2, 4^2), (4^2, 5^2), (5^2, 6^2), (6^2, 7^2), (7^2, 8^2), (8^2, 9^2), (9^2, 10^2), (10^2, 11^2), (11^2, 12^2), (12^2, 13^2), (13^2, 14^2), (14^2, 15^2), (15^2, 16^2), (12^2, 16^2), (16^2, 17^2), (17^2, 18^2), (18^2, 19^2), (17^2, 19^2), (19^2, 20^2), (20^2, 21^2), (21^2, 22^2), (22^2, 23^2), (23^2, 27^2), (9^2, 24^2), (24^2, 25^2), (25^2, 26^2), (26^2, 27^2), (27^2, 28^2), (28^2, 29^2), (29^2, 30^2), (30^2, 31^2), (31^2, 33^2), (30^2, 32^2), (32^2, 33^2), (33^2, 34^2) }
E_{s}	{ (6^{m}, 6^0), (6^{m}, 6^1), (6^{m}, 6^2), (9^{m}, 10^0), (9^{m}, 22^0), (9^{m}, 23^0), (9^{m}, 24^0), (9^{m}, 10^1), (9^{m}, 22^1), (9^{m}, 23^1), (9^{m}, 24^1), (9^{m}, 10^2), (9^{m}, 22^2), (9^{m}, 23^2), (9^{m}, 24^2), (18^{m}, 10^0), (18^{m}, 22^0), (18^{m}, 23^0), (18^{m}, 24^0), (18^{m}, 10^1), (18^{m}, 22^1), (18^{m}, 23^1), (18^{m}, 24^1), (18^{m}, 10^2), (18^{m}, 22^2), (18^{m}, 23^2), (18^{m}, 24^2), (22^{m}, 31^0), (22^{m}, 32^0), (22^{m}, 31^1), (22^{m}, 32^1), (22^{m}, 31^2), (22^{m}, 32^2), (24^{m}, 31^0), (24^{m}, 32^0), (24^{m}, 31^1), (24^{m}, 32^1), (24^{m}, 31^2), (24^{m}, 32^2), (6^0, 6^{m}), (6^0, 6^1), (6^0, 6^2), (19^0, 12^{m}), (19^0, 10^1), (19^0, 22^1), (19^0, 23^1), (19^0, 24^1), (19^0, 10^2), (19^0, 22^2), (19^0, 23^2), (19^0, 24^2), (23^0, 12^{m}), (23^0, 10^1), (23^0, 22^1), (23^0, 23^1), (23^0, 24^1), (23^0, 10^2), (23^0, 22^2), (23^0, 23^2), (23^0, 24^2), (25^0, 12^{m}), (25^0, 10^1), (25^0, 22^1), (25^0, 23^1), (25^0, 24^1), (25^0, 10^2), (25^0, 22^2), (25^0, 23^2), (25^0, 24^2), (28^0, 28^1), (28^0, 28^2), (31^0, 22^{m}), (31^0, 24^{m}), (31^0, 31^1), (31^0, 32^1), (31^0, 31^2), (31^0, 32^2), (32^0, 22^{m}), (32^0, 24^{m}), (32^0, 31^1), (32^0, 32^1), (32^0, 31^2), (32^0, 32^2), (6^1, 6^{m}), (6^1, 6^0), (6^1, 6^2), (19^1, 12^{m}), (19^1, 10^0), (19^1, 22^0), (19^1, 23^0), (19^1, 24^0), (19^1, 10^2), (19^1, 22^2), (19^1, 23^2), (19^1, 24^2), (23^1, 12^{m}), (23^1, 10^0), (23^1, 22^0), (23^1, 23^0), (23^1, 24^0), (23^1, 10^2), (23^1, 22^2), (23^1, 23^2), (23^1, 24^2), (25^1, 12^{m}), (25^1, 10^0), (25^1, 22^0), (25^1, 23^0), (25^1, 24^0), (25^1, 10^2), (25^1, 22^2), (25^1, 23^2), (25^1, 24^2), (28^1, 28^0), (28^1, 28^2), (31^1, 22^{m}), (31^1, 24^{m}), (31^1, 31^0), (31^1, 32^0), (31^1, 31^2), (31^1, 32^2), (32^1, 22^{m}), (32^1, 24^{m}), (32^1, 31^0), (32^1, 32^0), (32^1, 31^2), (32^1, 32^2), (6^2, 6^{m}), (6^2, 6^0), (6^2, 6^1), (19^2, 12^{m}), (19^2, 10^0), (19^2, 22^0), (19^2, 23^0), (19^2, 24^0), (19^2, 10^1), (19^2, 22^1), (19^2, 23^1), (19^2, 24^1), (23^2, 12^{m}), (23^2, 10^0), (23^2, 22^0), (23^2, 23^0), (23^2, 24^0), (23^2, 10^1), (23^2, 22^1), (23^2, 23^1), (23^2, 24^1), (25^2, 12^{m}), (25^2, 10^0), (25^2, 22^0), (25^2, 23^0), (25^2, 24^0), (25^2, 10^1), (25^2, 22^1), (25^2, 23^1), (25^2, 24^1), (28^2, 28^0), (28^2, 28^1), (31^2, 22^{m}), (31^2, 24^{m}), (31^2, 31^0), (31^2, 32^0), (31^2, 31^1), (31^2, 32^1), (32^2, 22^{m}), (32^2, 24^{m}), (32^2, 31^0), (32^2, 32^0), (32^2, 31^1), (32^2, 32^1) }
E	$E_i^p \cup E_{\mathrm{s}}$

DOI: 10.1002/cpe

Table III. Some elements required by the proposed testing criteria for the ring program with four processes (one master and three slaves).

Criteria	Required elements
all-nodes	1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, ..., 1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0, 8^0, 9^0, ..., 1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, ..., 1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, ...
all-nodes-std	2^m, 3^m, 4^m, 5^m, 6^m, 8^m, 9^m, ..., 2^0, 3^0, 4^0, 5^0, 6^0, 8^0, 10^0, 11^0, 13^0, ..., 2^1, 3^1, 4^1, 5^1, 6^1, 8^1, 10^1, ..., 2^2, 3^2, 4^2, 5^2, 6^2, 8^2, 10^2, 11^2, 13^2, ...
all-nodes-s	(6^m, 1, *'BSR'*), (9^m, 0, *'NS'*), (18^m, 0, *'NS'*), ..., (6^0, 1, *'BSR'*), (19^0, 0, *'NS'*), (23^0, 0, *'BSR'*), ..., (25^1, 0, *'BS'*), (28^1, 3, *'BSR'*), (31^1, 2, *'BSR'*), ..., (31^2, 2, *'BSR'*), (32^2, 2, *'BSR'*)
all-nodes-r	(6^m, 1, *'BSR'*), (12^m, 0, *'NR'*), (22^m, 2, *'BSR'*), ..., (6^0, 1, *'BSR'*), (10^0, 0, *'NR'*), (22^0, 0, *'BR'*), ..., (22^1, 0, *'BR'*), (23^1, 0, *'BSR'*), (24^1, 0, *'BR'*), ..., (31^2, 2, *'BSR'*), (32^2, 2, *'BSR'*)
all-nodes-nt	(13^m, *True*), (13^m, *False*), (11^0, *True*), (11^0, *False*), (11^1, *True*), (11^1, *False*), (11^2, *True*), (11^2, *False*)
all-edges-s	(6^m, 6^0), (6^m, 6^1), ..., (6^0, 6^m), (6^0, 6^1), ..., (19^0, 10^1), (19^0, 22^1), ..., (23^1, 12^m), (23^1, 10^0), ..., (32^2, 31^1), (32^2, 32^1)
all-edges	(1^m, 2^m), (2^m, 3^m), ..., (1^0, 2^0), (2^0, 3^0), ..., (1^1, 2^1), (2^1, 3^1), ..., (1^2, 2^2), (2^2, 3^2), ..., (6^m, 6^0), (6^m, 6^1), ..., (6^0, 6^m), (6^0, 6^1), ..., (19^0, 10^1), (19^0, 22^1), ... , (23^1, 12^m), (23^1, 10^0), ..., (32^2, 31^1), (32^2, 32^1)
all-defs	(7^m, (9^m, 10^0), *tkn*), (18^m, (21^m, 22^m), *min*), (23^0, (30^0, 31^0), *min*), (7^0, 8^0, *next*), (4^1, 7^1, *nproc*), (8^2, 25^2, *reqid*1), ...
all-defs-s	(7^m, (9^m, 10^0), 16^0, *tkn*, *tkn*), (18^1, (19^1, 24^2), (25^2, 22^0), *tkn*, *tkn*), (23^0, (28^0, 28^1), (30^1, 31^1), *min*, *min*), (24^2, (25^2, 22^0), 29^0, *tkn*, *tkn*), ...
all-c-uses	(22^0, 29^0, *tkn*), (24^1, 29^1, *tkn*), (3^1, 7^1, *my_rank*), (1^m, 7^m, *argv*), ...
all-p-uses	(12^m, (21^m, 22^m), *min*), (12^m, (21^m, 24^m), *min*), (11^1, (12^1, 13^1), *arrived*), ...
all-s-uses	(7^m, (9^m, 10^0), *tkn*), (7^m, (9^m, 10^1), *tkn*), (7^m, (9^m, 10^2), *tkn*), ..., (23^0, (28^0, 28^1), *min*), (23^0, (28^0, 28^2), *min*), (23^1, (28^1, 28^0), *min*), ...
all-s-c-uses	(7^m, (9^m, 10^0), 16^0, *tkn*, *tkn*), (7^m, (9^m, 10^1), 16^1, *tkn*, *tkn*), (7^m, (9^m, 10^2), 16^2, *tkn*, *tkn*), (18^0, (19^0, 24^1), 29^1, *tkn*, *tkn*), (24^1, (25^1, 24^2), 29^2, *tkn*, *tkn*), ...
all-s-p-uses	(16^0, (23^0, 12^m), (21^m, 24^m), *min*, *min*), (16^1, (23^1, 12^m), (21^m, 22^m), *min*, *min*), (23^0, (28^0, 28^2), (30^2, 31^2), *min*, *min*), (23^1, (28^1, 28^0), (30^0, 31^0), *min*, *min*), ...
all-nr-uses	(12^m, (21^m, 22^m), *min*), (10^0, 16^0, *tkn*), (10^1, 16^1, *tkn*), (10^2, 16^2, *tkn*), (10^0, (17^0, 18^0), *tkn*), (10^1, (17^1, 18^1), *tkn*), ...
all-ns-uses	(9^m, 12^m, *tkn*), (19^0, 22^0, *tkn*), (25^1, 29^1, *tkn*), ...

old value attributed to it at node 7^0 as a consequence of this synchronization fault. This fault can be revealed by the *all-nodes-nt* criterion, which shows the program behavior when the message arrives before and then after the non-blocking test primitive at node 11^0. To cover both required elements (11^0, *True*) and (11^0, *False*) from the *all-nodes-nt* criterion, the tester can choose the test case {0}. An unexpected output (minimum = 0) is obtained when covering (11^0, *False*), and it can be observed that the s-use association (7^m, (9^m, 10^0), *tkn*, *tkn*) is not covered because of the synchronization of processes and the lack of the *MPI_Wait* primitive. When the required element (11^0, *True*) is covered by running the test case {0}, the fault is not revealed. If the test input {1} is executed by the tester, the fault will not be revealed when covering both required elements

DOI: 10.1002/cpe

(11^0, *True*) and (11^0, *False*). However, the s-use association (7^m, (9^m, 10^0), *tkn*, *tkn*) is not covered when executing (11^0, *False*), and then the tester can conclude that the program has a fault related to non-determinism. Analyzing the execution trace, he or she can observe which edge(s) has performed an unexpected synchronization. The ValiPar tool provides the necessary support in this case, offering controlled execution and allowing the analysis of the execution trace.

A similar observability fault occurs when the definition *min* = *tkn* * *tkn* at node 16^0 is moved to come immediately after node 11^0. In this new scenario, the value of *min* is correctly updated if and only if the message reaches p^0 before the execution of the non-blocking testing at node 11^0; otherwise, the value attributed to *min* is the one defined earlier at node 7^0 to *tkn*, that is, $\{-1\}$. Again, this fault can be revealed using the *all-nodes-nt* criterion, when trying to cover its required element (11^0, *False*) with the test input {0}. This execution requires controlled execution support.

Some examples of mistakes in the slave source code capable of producing locking faults are as follows: replacing the operator '%' for '/' at node 7, replacing the *MPI_COMM_WORLD* parameter for *icomm* at node 8, and changing the *MPI_ANY_TAG* parameter for the constant 4. These mistakes will lock the slave execution at nodes 22 and 24 (blocking receivers), waiting forever for messages sent to another rank, context, or tag, respectively. The *all-edges-s* criterion can help the tester to find these mistakes because the expected edges (19^0, 24^1), (25^1, 24^2), and (25^2, 22^0) are not covered before the application locks.

Deadlock is a special type of locking fault and represents a classical problem in parallel programs [25]. Ideally, it should be detected before execution of the parallel program. It is not the focus of the testing criteria proposed in this work; nonetheless, the information extracted of the parallel programs during the application of the coverage criteria may be used to help detect deadlocks.

5.2. *Comparing the new model and criterion costs*

In this section, we present the application costs of our new test model and criteria for message-passing concurrent programs, comparing them to our previous work [5]. We consider the set sizes generated for the test model and the number of required elements for each criterion. Two programs are considered in this analysis: the ring program described previously and also the Jacobi program described in the following.

The Jacobi program implements the iterative method of the Gauss–Jacobi for solving a linear system of equations. It is composed of 10 concurrent processes (one master and nine slaves), where the master p^m distributes chunks of the system for the slaves that determine together whether the system can converge or not. If the system is capable of converging, the master initiates the iterative part of the code to solve the system with iterations using the slaves. The master analyzes the stop criterion at each iteration end, and when it is reached, it prints the final result. There are around 600 nodes in the PCFG generated for this application, 135 sender nodes, and 144 receiver nodes. This example is representative of the following classes of communication primitives: point to point, collective, blocking, and non-blocking. It also considers handler groups and the single-program, multiple-data paradigm.

Although a broader range of studies would be required to achieve a statistically significant result, these results provide preliminary evidence of the usefulness of the new test model and criteria proposed in this paper.

The data presented in this section were captured using different test inputs (two for the ring program and four for the Jacobi program) and also controlled execution support. The required elements covered by each criterion were detected, and the elements not covered were classified as infeasible. Table IV presents the number of items inserted in each set belonging to the new and previous models. By analyzing the results, we observe that the new model is able to represent the new features from the message-passing concurrent programs and also can reduce significantly the size of some important sets, such as E_s and R_i^p. This reduction is due to the classification of the communication primitives, creating clusters of primitives able to match with each other. The replacement of both sets N_s and N_r by N_{sync} avoids the replication of nodes acting as senders and

DOI: 10.1002/cpe

receivers, reducing by 33% and 39% the amount of items used to represent the same quantity of send and receive primitives existent in the ring and Jacobi programs, respectively.

Table V presents the required elements generated/covered for each criterion. The difference between them determines the infeasible required elements. We can observe, from these results, that the four new criteria proposed are applicable and can result in a reduced size when compared with the other criteria already proposed, such as *all-edge-s*.

The amount of required elements was reduced significantly in relation to our previous test model, by removing the generation of infeasible elements. Considering the ring program, the reduction was 68% for *all-edges-s*, 62% for *all-s-uses*, 60% for *all-s-c-uses*, and 71% for *all-s-p-uses*. The

Table IV. Total of elements represented in the sets for the ring and Jacobi programs.

	Ring program		Jacobi program	
Sets	New model	Original model	New model	Original model
Prog	4	4	10	10
N	129	129	612	612
N_{std}	92	NA	322	NA
N_{sync}	36	NA	171	NA
N_{s}	NA	26	NA	135
N_{r}	NA	28	NA	144
$N_{\text{recv\_init}}$	1	NA	0	NA
$N_{\text{send\_init}}$	3	NA	0	NA
N_{start}	7	NA	0	NA
N_{wait}	12	NA	3	NA
N_{test}	4	NA	3	NA
$N_{\text{get\_status}}$	0	NA	0	NA
N_{iprobe}	0	NA	0	NA
E_i^p	139	139	734	734
E_{s}	171	540	1332	18,200
E	310	679	2066	18,934

NA, not applicable.

Table V. Total of required elements using the proposed testing criteria for the ring and Jacobi programs (required elements/feasible elements).

	Ring program			Jacobi program		
Criteria	New model	Original model	Reduction (%)	New model	Original model	Reduction (%)
all-nodes	129/129	129/129	0.0	612/612	612/612	0.0
all-nodes-std	92/92	NA	—	322/322	NA	—
all-nodes-s	26/26	26/26	0.0	135/135	135/135	0.0
all-nodes-r	28/28	28/28	0.0	144/144	144/144	0.0
all-nodes-nt	8/8	NA	—	6/6	NA	—
all-edges-s	171/26	540/26	68.3	1332/61	18,200/61	92.7
all-edges	310/165	679/165	54.3	2066/795	18,934/795	89.1
all-defs	68/68	68/68	0.0	799/799	799/799	0.0
all-defs-s	68/68	68/68	0.0	799/799	799/799	0.0
all-c-uses	59/59	59/59	0.0	538/538	538/538	0.0
all-p-uses	34/34	34/34	0.0	199/199	199/199	0.0
all-s-uses	162/33	429/33	62.2	1440/64	18632/64	92.3
all-s-c-uses	132/90	327/90	59.6	426/44	1647/44	74.1
all-s-p-uses	78/14	270/14	71.1	298/22	639/22	53.4
all-nr-uses	10/10	NA	—	2/2	NA	—
all-ns-uses	10/10	NA	—	2/2	NA	—

NA, not applicable.

DOI: 10.1002/cpe

reduction for the Jacobi program was 93% for *all-edges-s*, 92% for *all-s-uses*, 74% for *all-s-c-uses*, and 53% for *all-s-p-uses*. The reduction of 16,868 infeasible sync-edges in the Jacobi program is an excellent result to decrease the testing activity cost, compared with a total of 18,200 sync-edges produced earlier by our previous test model. These reductions mean that the tester will not be required to analyze 927 elements of the ring program and 35,622 elements of the Jacobi program during the testing activity.

It is important to note that the reduction of required elements is directly related to the use of communication primitives belonging to distinct clusters, such as point to point and collectives. The use of distinct types of communication primitives is a style of programming expected when developing concurrent programs in MPI. Indeed, this practice improves developer productivity, reduces lines of code (potentially avoiding more mistakes), and optimizes overall performance. However, if the source code uses just one type of primitive (e.g., point to point), this reduction may not always be achieved. This limitation can be overcome by grouping primitives in different ways, such as intra-communicators/inter-communicators, source/target/tags, and control/data flows. This approach requires extra effort (and thus testing cost) to generate the N_{sync} set; but it can potentially detect infeasible matches among point-to-point primitives, specific collective primitives, and so on. The case studies demonstrate that the reduction of testing cost thus achieved fully justifies the extra testing effort required. Considering this scenario, it is possible to note that our proposal provides increased model flexibility, improves the test activity quality, and, at same time, reduces the cost of the testing activity.

In spite of the considerable number of required elements, some effort is necessary to identify infeasible elements. A good strategy, as adopted in our previous test model, is to run test cases with the controlled execution support and analyze the required elements to decide about the feasibility only when the addition of new test cases does not contribute to improving the coverage. In this last case, paths are identified to cover the remaining elements, and if possible, particular test cases can be generated. A possible complementary strategy that can be used here is the work [19] discussed in Section 2, which uses reachability testing to guide the choice of required elements by discarding infeasible elements. This strategy has produced significant reduction of the costs during the testing activity application.

6. RELATED WORK

Research into concurrent software testing is a very active area with many relevant published papers. They can be classified in several different ways, considering for example static or dynamic approaches and shared-memory and/or message-passing paradigms and also considering the language and/or standard used. The published papers are distributed over a number of discrete testing topics, such as failure injection [26], formal verification [27–33], static analysis [34–36], testing-driven development [37–39], controlled execution [40–46], mutation testing [47–49], model checking [50–53], model-based testing [54, 55], structural testing [56–72], symbolic analysis [73, 74], search-based testing [75, 76], interleaving coverage testing [77], probabilistic concurrency testing [78], reachability testing [20, 79–91], and test case generation [92–94].

We highlight just some of the relevant related research here, given the scope of this paper. A larger set of references can be found in the systematic review in [95].

Yang and Chung [59] proposed a static analysis technique to identify whether a given concurrent path is traversable in some execution. The technique is based on the analysis of the possible execution order of rendezvous statements within the given concurrent path. The execution order of rendezvous statements in this concurrent path is used to define a precede relation. In addition, several precedence rules and an algorithm are proposed to derive precede relations defined in a concurrent path. From the precede relations and the semantics of Ada, decision rules are considered to determine statically infeasible paths. This approach does not generate the combinatorial explosion problem that occurs in the reachability analysis.

Taylor *et al*. [96] proposed a set of structural coverage criteria for concurrent programs based on the notion of concurrent states and on the concurrency graph. Five criteria are defined:

DOI: 10.1002/cpe

all-concurrency-paths, *all-proper-cc-histories*, *all-edges-between-cc-states*, *all-cc-states*, and *all-possible-rendezvous*. The hierarchy among these criteria is analyzed. They stressed that every approach based on reachability analysis would be limited in practice by state space explosion. They mentioned some alternatives to overcome the associated constraints.

In the same vein as Taylor *et al*., Chung *et al*. [97] proposed four testing criteria for Ada programs: *all-entry-call*, *all-possible-entry-acceptance*, *all-entry-call-permutation*, and *all-entry-call-dependency-permutation*. These criteria focus on the rendezvous among tasks. They also presented a hierarchy among these criteria.

Yang *et al*. [98, 99] extended the data flow criteria [2] to shared-memory parallel programs. The parallel program model used consists of multiple threads of control that can be executed simultaneously. A *parallel program flow graph* is constructed and is traversed to obtain the paths, variable definitions, and uses. All paths that have definition and use of variables related with parallelism of threads constitute test requirements to be exercised. The *Della Pasta Tool (Delaware Parallel Software Testing Aid)* automates their approach. The authors present the foundations and theoretical results for structural testing of parallel programs, with a definition of the *all-du-path* and *all-uses* criteria for shared-memory programs. This work inspired the test model definition for message-passing parallel programs, described in Section 2.

Yuan [100] defined an extension of CFG for BPEL programs, creating a BPEL flow graph. Test data and concurrent test paths are generated by traversing the BPEL flow graph model. In this way, complete test cases are generated through a combination of test paths and input data. This work focuses on BPEL test case generation and considers BPEL special features.

Kojima *et al*. [63] used the *All-Concurrent-Paths* criterion on a concurrent module flow graph to execute structural testing of embedded concurrent software present in different devices, such as high-definition TVs, recorders, and mobile phones. To overcome the very large quantity of test cases required when the number of elements increases, the authors proposed to suppress test cases considering the relation happens-before between local blocks (those that do not include operating shared resources) and focus the testing activity in the external operation blocks instead (i.e., those that operate on shared resources). This approach does not directly address the large quantity of synchronization pairs that occur in the static analysis.

Kojima *et al*. [36] presented a static analysis to detect message-passing errors in Erlang, considering dynamic process creation and communication based on asynchronous message passing. The proposed analysis is carried out by the *dialyzer*, a software tool that aims to balance soundness and completeness by using reduced CFGs and removing sync-edges given Erlang programming characteristics. In contrast to our proposal, the source code cannot be entirely covered because of the approach adopted for reducing the CFGs.

Humphrey *et al*. [101] presented a plug-in called Graphical Explorer of MPI programs, used by the dynamic verifier called the *in situ* partial (ISP) order. The ISP is able to search the execution space of an MPI program dynamically to detect a set of bugs related to deadlocks, resource leaks (such as MPI object leaks), and violations of C assertions placed in the code. The ISP is a dynamic tool able to exercise relevant interleaving in MPI programs and requires no model building. The Graphical Explorer of MPI programs plug-in acts as a link between the ISP and the Eclipse Foundation's Parallel Tools Platform. In contrast to our work, this paper presents a fully dynamic approach, designed to detect a specific set of errors.

Edelstein *et al*. [102, 103] presented a multithreaded bug detection architecture called *ConTest* for Java programs. This architecture combines a replay algorithm with a seeding technique, where the coverage is specific to race conditions. This seeding technique inserts *sleep* statements into the program, on the basis of shared-memory accesses and synchronization events. Heuristics are used to decide when a *sleep* statement must be activated. The replay algorithm is used to re-execute a test when race conditions are detected, ensuring that all accesses in a race will be executed. The focus of this work is the non-determinism problem and not specifically code coverage and testing criteria.

Sherman *et al.* [57] investigated the use of saturation-based testing for Java concurrent programs. A saturation point happens when additional testing effort is unlikely to provide additional coverage of the source code. The authors applied the saturation point concept to complete the testing activity

DOI: 10.1002/cpe

rather than consider the coverage rate. Contrasting with our work, this approach abandons the target of achieving 100% test coverage, aiming instead to strike a balance between cost and fault detection for concurrent programs.

Wong *et al*. [88] proposed a set of methods to generate test sequences for the structural testing of concurrent programs. The reachability graph is used to represent concurrent programs and to select test sequences using the *all-node* and *all-edge*. The methods aim to generate a small test sequence set that will cover all the nodes and the edges in a reachability graph. For this, the methods provide information about which parts of the program should be covered first effectively to increase the coverage of these criteria. The authors stressed that the major advantage of the reachability graph is that only feasible paths will be generated. However, the authors did not explain how to generate the reachability graph from the concurrent program or how to deal with the state space explosion.

Lei and Carver [20] proposed a method (based on reachability testing) to obtain all of the executable synchronizations of a concurrent program (from a given execution of the program) in a way that reduces the number of redundant synchronizations. As this method uses dynamic information, only feasible synchronizations are generated, which is a considerable advantage. However, a difficulty with this method is the high number of possible combinations of synchronization that are generated. For complex programs, this number is extremely high, which limits the practical application of this approach. In [81], Carver and Lei proposed a distributed reachability testing algorithm, allowing different test sequences to be executed concurrently. This algorithm reduces the time to execute the synchronizations, but the authors did not comment about the effort necessary to analyze the results from these executions. The method described in [20] is essentially complementary to our approach. However, the authors did not address how to select the test case that will be used for the initial run, whereas we make use of the static analysis of the program to select the optimum test cases in advance.

7. CONCLUDING REMARKS AND FUTURE WORK

Testing parallel programs is a major research challenge that has been widely investigated by many researchers. This paper contributes to this research effort by extending the structural test model proposed in [5]. We address different features of message-passing concurrent programs not previously considered in our earlier work: non-blocking receives and sends, persistent and collective communication primitives, different contexts of communication (inter-communicators), nodes with sender/receiver behavior in the same execution, and sends that are not able to reach all of the receives. The main contribution of our work is a proposed flexible test model that provides improved coverage of message-passing programs, together with a significantly reduced testing activity cost.

The flexibility of our test model has been improved in several different ways, mainly because now it is possible to generalize sends and receives in the same set N_{sync}, where N_{sync} can be configured to represent logical sub-groups (or clusters) of primitives able to interact with each other. Clustering primitives reduces significantly the amount of infeasible sync-edges and, consequently, the amount of s-use, s-c-use, and s-p-use associations. This relatively straightforward change is nevertheless very important because of the reduction in costs of detecting infeasible elements automatically. Indeed, the large amount of infeasible elements that must still be analyzed manually by the tester represents a very high cost in the static approach. The extension proposed in this paper contributes significantly in this direction by creating a test model able to support the primitive cluster concept.

We propose four new testing criteria in addition to the ones already provided. They improve overall testing activity quality by acting as a guideline for test case generation and also by establishing a metric to determine whether the testing activity is completed or not. These new criteria concentrate on message-passing standard primitives, non-blocking checking of the communication primitive status, and the use of buffers after non-blocking communication primitives.

The case study results demonstrate that the sets of elements extracted from the source code are effective in representing the features considered by the new model. Besides raising the amount of features represented by the test model, this extended model contributes to reducing the

DOI: 10.1002/cpe

testing costs. The number of sync-edges (E_s) has been reduced up to 93%, by eliminating infeasible edges between unmatchable primitives. Considering these reductions in E_s, the amount of required elements generated for the testing criteria related to the communication also was consequently reduced. Considering our case studies, these reductions mean that the tester will not be required to analyze 927 elements of the ring program and 35,622 elements of the Jacobi program during the testing activity.

Future work will be directed to the following lines of research: (i) experimenting on clustering communication primitives according to other factors, such as intra-communicators/inter-communicators, source/target/tags, and control/data flows; (ii) applying this new test model to a one-sided communication model by taking into account the concepts proposed in [6]; (iii) developing experiments to refine and evaluate structural testing criteria related to inter-process communication; and (iv) developing mechanisms to support automatic test case generation.

ACKNOWLEDGEMENT

This work is sponsored by FAPESP, INCT/SEC, and CAPES (2010/02839-0, 2008/57870-9, 573963/2008-8, and 1191/10-1).

REFERENCES

1. Grama A, Gupta A, Karypis G, Kumar V. *Introduction to Parallel Computing*, 2nd edn. Addison Wesley: Harlow, 2003.
2. Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transaction Software Engineering* 1985; **11**(4):367–375.
3. Gropp W, Lusk E, Skjellum A. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press: Cambridge, 1999.
4. Forum MPI. *MPI: A Message-Passing Interface Standard—Version 2.2*. University of Tennessee: Knoxville, TN, 2009.
5. Souza SRS, Vergilio SR, Souza PSL, Simao AS, Hausen AC. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience* 2008; **20**:1893–1916.
6. Sarmanho FS, Souza PSL, Souza SRS, Simao AS. Structural testing for semaphore-based multithread programs. In *International Conference on Computational Science*, Vol. 5101, Bubak M, van Albada G, Dongarra J, Sloot P (eds), Lecture Notes in Computer Science. Springer: Heidelberg, 2008; 337–346.
7. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain RH, Daniel DJ, Graham RL, Woodall TS. Open MPI: goals, concept, and design of a next generation MPI implementation. *11th European PVM/MPI Users' Group Meeting*, 2004; 97–104.
8. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI Message Passing Interface standard. *Parallel Computing* 1996; **22**(6):789–828.
9. Gropp WD, Lusk E. *User's Guide for MPICH, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory: Argonne, IL, 1996. ANL-96/6.
10. Franke H, Wu CE, Riviere M, Pattnaik P, Snir M. MPI programming environment for SP1/SP2. *15th International Conference on Distributed Computing Systems*, Vancouver, British Columbia, Canada, 1995; 127–135.
11. Frankl FG, Weyuker EJ. Data flow testing in the presence of unexecutable paths. *Workshop on Software Testing*, New York, USA, 1986; 4–13.
12. Carver RH, Tai KC. Replay and testing for concurrent programs. *IEEE Software* 1991; **8**(2):86–74.
13. Vergilio SR, Souza SRS, Souza PSL. Coverage testing criteria for message-passing parallel programs. *6th IEEE Latin-American Test Workshop (LATW 2005)*, Salvador, Bahia, Brazil, 2005; 161–166.
14. Souza SRS, Vergilio SR, Souza PSL, Simão AS, Bliscosque TG, Lima AM, Hausen AC. ValiPar: a testing tool for message-passing parallel programs. *International Conference on Software Knowledge and Software Engineering (SEKE 2005)*, Taipei, Taiwan, 2005; 386–391.
15. Souza PSL, Sawabe E, Simao AS, Vergilio SR, Souza SRS. ValiPVM—a graphical tool for structural testing of PVM programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Vol. 5205, Lastovetsky A, Kechadi T, Dongarra J (eds), Lecture Notes in Computer Science. Springer: Berlin, 2008; 257–264.
16. Hausen AC, Vergílio SR, Souza SRS, Souza PSL, Simão AS. A tool for structural testing of MPI programs. *8th IEEE Latin-American Test Workshop (LATW 2007)*, Cuzco, Peru, 2007; 1–6.
17. Endo AT, Simao AS, Souza SRS, Souza PSL. Web services composition testing: a strategy based on structural testing of parallel programs. *Testing: Academic Industrial Conference—Practice and Research Techniques (TAIC-PART 2008)*, Windsor, UK, 2008; 3–12.
18. Simao AS, Vincenzi AMR, Maldonado JC, Santana ACL. A language for the description of program instrumentation and the automatic generation of instrumenters. *CLEI Electronic Journal* 2003; **6**(1):1–23.

DOI: 10.1002/cpe

19. Souza SRS, Souza PSL, Machado MCC, Camillo MS, Simao AS, Zaluska E. Using coverage and reachability testing to improve concurrent program testing quality. *23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, Eden Roc Renaissance Miami Beach, USA, 2011; 207–212.
20. Lei Y, Carver RH. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering* 2006; **32**(6):382–403.
21. Mohnen M. A graph free approach to data flow analysis. In *Compiler Construction*, Vol. 2304, Horspool R (ed.), Lecture Notes in Computer Science. Springer: Berlin, 2002; 185–213.
22. Khedker UP, Sanyal A, Karkare B. *Data Flow Analysis: Theory and Practice*. CRC Press (Taylor and Francis Group): Florence, KY, 2009.
23. Howden WE. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering* 1976; **2**:208–215.
24. Krawczyk H, Wiszniewski B. Classification of software defects in parallel programs. *Technical Report 2*, Faculty of Electronics, Technical University of Gdansk, Poland, 1994.
25. Tanenbaum AS. *Modern Operating Systems*, 2nd edn. Prentice Hall: Upper Saddle River, NJ, 2001.
26. Artho C, Biere A, Honiden S. Enforcer—efficient failure injection. In *Formal Methods*, Vol. 4085, Misra A, Nipkow T, Sekerinski E (eds), Lecture Notes in Computer Science. Springer: Berlin, 2006; 412–427.
27. Kavi KM, Moshtaghi A, Chen DJ. Modeling multithreaded applications using Petri nets. *International Journal of Parallel Programming* 2002; **30**(5):353–371.
28. Chen F, Roşu G. Parametric and sliced causality. In *Computer Aided Verification*, Vol. 4590, Damm W, Hermanns H (eds), Lecture Notes in Computer Science. Heidelberg: Berlin, 2007; 240–253.
29. Yang Z, Sakallah K. Trace-driven verification of multithreaded programs. In *Formal Methods and Sofware Engineering*, Vol. 6447, Song Dong J, Zhu H (eds), Lecture Notes in Computer Science. Heidelberg: Berlin, 2010; 404–419.
30. Wu M, Zhou B, Shi W. A self-adaptive test framework for concurrent programs. *International Conference on Management of Emergent Digital Ecosystems (MEDES 2009)*, Lyon, France, 2009; 456–457.
31. Rakamarić Z. Storm: static unit checking of concurrent programs. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*. ACM: New York, NY, 2010; 519–520.
32. Wood BP, Sampson A, Ceze L, Grossman D. Composable specifications for structured shared-memory communication. *ACM SIGPLAN Notices (OOPSLA 2010)* 2010; **45**:140–159.
33. Xu K, Liang D. Formally defining a graphical language for monitoring and checking object interactions. In *Model Driven Engineering Languages and Systems*, Vol. 4735, Engels G, Opdyke B, Schmidt D, Weil F (eds), Lecture Notes in Computer Science. Springer: Heidelberg, 2007; 620–634.
34. Chen Q, Wang L, Yang Z, Stoller SD. HAVE: detecting atomicity violations via integrated dynamic and static analysis. In *Fundamental Approaches to Software Engineering*, Vol. 5503, Chechik M, Wirsing M (eds), Lecture Notes in Computer Science. Springer: Berlin, 2009; 425–439.
35. Chen J. Guided testing of concurrent programs using value schedules. *PhD Thesis*, University of Waterloo, 2009.
36. Christakis M, Sagonas K. Detection of asynchronous message passing errors using static analysis. In *Practical Aspects of Declarative Languages*, Vol. 6539, Rocha R, Launchbury J (eds), Lecture Notes in Computer Science. Springer: Berlin, 2011; 5–18.
37. Dantas A, Brasileiro F, Cirne W. Improving automated testing of multi-threaded software. *1st International Conference on Software Testing, Verification and Validation (ICST 2008)*, Lillehammer, Norway, 2008; 521–524.
38. Ricken M, Cartwright R. ConcJUnit: unit testing for concurrent programs. *7th International Conference on Principles and Practice of Programming in Java (PPPJ 2009)*, Calgary, Alberta, Canada, 2009; 129–132.
39. Jagannath V, Gligoric M, Jin D, Rosu G, Marinov D. IMUnit: improved multithreaded unit testing. In *3rd International Workshop on Multicore Software Engineering*, IWMSE ‘10. ACM: New York, NY, 2010; 48–49.
40. Ball T, Burckhardt S, Coons KE, Musuvathi M, Qadeer S. Preemption sealing for efficient concurrency testing. *Technical Report*, Microsoft, 2009.
41. Dantas A. Improving developers’ confidence in test results of multi-threaded systems: avoiding early and late assertions. *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, 2008; 899–900.
42. Dantas A, Gaudencio M, Brasileiro F, Cirne W. Obtaining trustworthy test results in multi-threaded systems. *Technical Report*, Federal University of Campina Grande, 2007.
43. Emmi M, Qadeer S, RakamariA Z. Delay-bounded scheduling. *ACM SIGPLAN Notices (POPL 2011)* 2011; **46**(1):411–422.
44. Yu J, Narayanasamy S. A case for an interleaving constrained shared-memory multi-processor. In *36th Annual International Symposium on Computer Architecture (ISCA 2009)*. ACM: New York, USA, 2009; 325–336.
45. Kamil A, Yelick K. Enforcing textual alignment of collectives using dynamic checks. In *Languages and Compilers for Parallel Computing*, Vol. 5898, Gao GR, Pollock LL, Cavazos J, Li X (eds), Lecture Notes in Computer Science. Springer: Berlin, 2010; 368–382.
46. Rungta N, Mercer EG. A meta heuristic for effectively detecting concurrency errors. In *Hardware and Sofware: Verification and Testing*, Vol. 5394, Chockler H, Hu AJ (eds), Lecture Notes in Computer Science. Springer: Heidelberg, 2009; 23–37.

DOI: 10.1002/cpe

47. Gligoric M, Jagannath V, Marinov D. MuTMut: efficient exploration for mutation testing of multithreaded code. *Third International Conference on Software Testing, Verification and Validation (ICST)*, Paris, France, 2010; 55–64.
48. Sen A, Abadir MS. Coverage metrics for verification of concurrent systemic designs using mutation testing. *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Anaheim, CA, United States, 2010; 75–81.
49. Jagannath V, Gligoric M, Lauterburg S, Marinov D, Agha G. Mutation operators for actor systems. *3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010)*, Paris, France, 2010; 157–162.
50. Yang Y, Chen X, Gopalakrishnan G. Inspect: a runtime model checker for multithreaded C programs. *Technical Report*, University of Utah, 2008.
51. Li J, Hei D, Yan L. Correctness analysis based on testing and checking for OpenMP programs. *Chinagrid Annual Conference (ChinaGrid 09)*, Beijing, China, 2009; 210–215.
52. Musuvathi M, Qadeer S. Fair stateless model checking. *Programming Language Design and Implementation (PLDI 08)*, Tucson, Arizona, 2008; 362–371.
53. Yang Z, Sakallah K. SMT-based symbolic model checking for multi-threaded programs. *Exploiting Concurrency Efficiently and Correctly Workshop*, Princeton, NJ, 2008.
54. Aichernig B, Griesmayer A, Schlatte R, Stam A. Modeling and testing multi-threaded asynchronous systems with Creol. *Electronic Notes in Theoretical Computer Science* 2009; **243**:3–14.
55. Aichernig BK, Griesmayer A, Johnsen EB, Schlatte R, Stam A. Conformance testing of distributed concurrent systems with executable designs. In *Formal Methods for Components and Objects*, Vol. 5751, de Boer FS, Bonsangue MM, Madelaine E (eds), Lecture Notes in Computer Science. Springer: Berlin, 2009; 61–81.
56. Takahashi J, Kojima H, Furukawa Z. Coverage based testing for concurrent software. *28th International Conference on Distributed Computing Systems Workshops (ICDCS 2008)*, Beijing, China, 2008; 533–538.
57. Sherman E, Dwyer MB, Elbaum S. Saturation-based testing of concurrent programs. In *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*. ACM: New York, USA, 2009; 53–62.
58. Yang RD, Chung CG. Path analysis testing of concurrent programs. *Information and Software Technology* 1992; **34**(1):43–56.
59. Yang RD, Chung CG. The analysis of infeasible concurrent paths of concurrent Ada programs. *Fourteenth Annual International Computer Software and Applications Conference (COMPSAC 1990)*, Chicago, Illinois, 1990; 424–429.
60. Yang RD, Chung CG. A path analysis approach to concurrent program testing. *International Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, 1990; 425–432.
61. Koppol PV, Tai KC. An incremental approach to structural testing of concurrent software. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1996)*. ACM: New York, USA, 1996; 14–23.
62. Koppol PV, Carver RH, Tai KC. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering* 2002; **28**(6):607–623.
63. Kojima H, Kakuda Y, Takahashi J, Ohta T. A model for concurrent states and its coverage criteria. *International Symposium on Autonomous Decentralized Systems (ISADS 2009)*, Athens, Greece, 2009; 1–6.
64. Krawczyk H, Wiszniewski B. A method for determining testing scenarios for parallel and distributed software. *Technical Report*, Technical University of Gdansk, Poland, 1996.
65. Wang YH, Chung CM, Shih TK, Keh HC, Lin WC. Software testing and metrics for concurrent computation through task decomposition. *IEEE International Conference on Intelligent Processing Systems (ICIPS 1997)*, Vol. 2, 1997; 1857–1861.
66. Shih TK, Chung CM, Wang YH, Kuo YF, Lin WC. Software testing and metrics for concurrent computation. *Asia-Pacific Software Engineering Conference*, Seoul, South Korea, 1996; 336–344.
67. Katayama T, Itoh E, Furukawa Z, Ushijima K. Test-case generation for concurrent programs with the testing criteria using interaction sequences. *Asia-Pacific Software Engineering Conference (APSEC 1999)*, Takamatsu, Japan, 1999; 590–597.
68. Katayama T, Furukawa Z, Ushijima K. Test-case generation method for concurrent programs including task-types. *Asia-Pacific Software Engineering Conference and International Computer Science Conference (APSEC/ICSC)*, Clear Water Bay, Hong Kong, 1997; 485–494.
69. Katayama T, Furukawa Z, Ushijima K. Event interactions graph for test-case generations of concurrent programs. *Asia-Pacific Software Engineering Conference*, Brisbane, Queensland, Australia, 1995; 29–37.
70. Katayama T, Furukawa Z, Ushijima K. A method for structural testing of Ada concurrent programs using the event interactions graph. *Asia-Pacific Software Engineering Conference*, Seoul, South Korea, 1996; 355–364.
71. Katayama T, Furukawa Z, Ushijima K. Design and implementation of test-case generation for concurrent programs. *Asia-Pacific Software Engineering Conference*, Taipei, Taiwan, ROC, 1998; 262–269.
72. Liang Y, Li S, Zhang H, Han C. Timing-sequence testing of parallel programs. *Journal of Computer Science and Technology* 2000; **15**(1):94–95.
73. Kundu S, Ganai MK, Wang C. CONTESSA: concurrency testing augmented with symbolic analysis. In *Computer Aided Verification*, Vol. 6174, Touili T, Cook B, Jackson P (eds). Springer: Heidelberg, 2010; 127–131.

DOI: 10.1002/cpe

74. Rungta N, Mercer EG, Visser W. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Model Checking Software*, Vol. 5578, Păsăreanu CS (ed.), Lecture Notes in Computer Science. Springer: Heidelberg, 2009; 174–191.
75. Krena B, Letko Z, Vojnar T, Ur S. A platform for search-based testing of concurrent software. *International Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2010)*, Trento, Italy, 2010; 48–58.
76. McMinn P. Search-based failure discovery using testability transformations to generate pseudo-oracles. *11th Annual Genetic and Evolutionary Computation Conference (GECCO-2009)*, Montreal, Canada, 2009; 1689–1696.
77. Lu S, Jiang W, Zhou Y. A study of interleaving coverage criteria. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*. ACM: New York, USA, 2007; 533–536.
78. Burckhardt S, Kothari P, Musuvathi M, Nagarakatte S. A randomized scheduler with probabilistic guarantees of finding bugs. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, 2010; 167–178.
79. Lei Y, Carver RH, Kacker R, Kung D. A combinatorial testing strategy for concurrent programs. *Software Testing Verification and Reliability* 2007; **17**(4):207–225.
80. Carver RH, Lei Y. A general model for reachability testing of concurrent programs. *Lecture Notes in Computer Science* 2004; **3308**:76–98.
81. Carver RH, Lei Y. Distributed reachability testing of concurrent programs. *Concurrency and Computation: Practice and Experience* 2010; **22**(18):2445–2466.
82. Lei Y, Carver R. Reachability testing of semaphore-based programs. *28th International Computer Software and Applications Conference (COMPSAC 2004)*, Vol. 1, Hong Kong, China, 2004; 312–317.
83. Hwang GH, Tai K, Huang T. Reachability testing: an approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering* 1995; **5**:493–510.
84. Li SQ, Chen HY, Sun YX. A framework of reachability testing for Java multithread programs. *IEEE International Conference on Systems, Man and Cybernetics*, Vol. 3, 2004; 2730–2734.
85. Wong WE, Lei Y. Reachability graph-based test sequence generation for concurrent programs. *International Journal of Software Engineering and Knowledge Engineering* 2008; **18**(6):803–822.
86. Carver RH, Lei Y. A stateful approach to testing monitors in multithreaded programs. *IEEE 12th International Symposium on High-Assurance Systems Engineering (HASE)*, San Jose, CA, 2010; 54–63.
87. Gong X, Wang Y, Zhou Y, Li B. On testing multi-threaded Java programs. *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, Vol. 1, Qingdao, China, 2007; 702–706.
88. Wong WE, Lei Y, Ma X. Effective generation of test sequences for structural testing of concurrent programs. *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Shanghai, China, 2005; 539–548.
89. Chen HY, Sun YX, Tse TH. A scheme for dynamic detection of concurrent execution of object-oriented software. *IEEE International Conference on Systems, Man and Cybernetics*, Vol. 5, Washington, D.C., USA., 2003; 4828–4833.
90. Pu F, Xu HY. A feasible strategy for reachability testing of internet-based concurrent programs. *IEEE International Conference on Networking, Sensing and Control (ICNSC 2008)*, Hainan, China, 2008; 1559–1564.
91. Wei W, Wu Y, Lejun Z, Lin G. Testing path generation algorithm with network performance constraints for nondeterministic parallel programs. *Seventh International Conference on Web-Age Information Management Workshops (WAIM 2006)*, Hong Kong, China, 2006; 6.
92. Ding Z, Zhang K, Hu J. A rigorous approach towards test case generation. *Information Sciences* 2008; **178**(21):4057–4079.
93. Tan RP, Nagpal P, Miller S. Automated black box testing tool for a parallel programming library. *2nd International Conference on Software Testing, Verification, and Validation (ICST 2009)*, Denver, Colorado, USA, 2009; 307–316.
94. Xiaoan B, Na Z, Zuohua D. Test case generation of concurrent programs based on event graph. *5th International Joint Conference on INC, IMS, and IDC (NCM 2009)*, Seoul, Korea, 2009; 143–149.
95. Souza SRS, Brito MAS, Silva RA, Souza PSL, Zaluska E. Research in concurrent software testing: a systematic review. *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2011), in Conjunction with International Symposium on Software Testing and Analysis (ISSTA 2011)*, Toronto, ON, Canada, 2011; 1–5.
96. Taylor RN, Levine DL, Kelly C. Structural testing of concurrent programs. *IEEE Transaction on Software Engineering* 1992; **18**(3):206–215.
97. Chung C-M, Shih TK, Wang Y-H, Lin W-C, Kou Y-F. Task decomposition testing and metrics for concurrent programs. *Fifth International Symposium on Software Reliability Engineering (ISSRE 1996)*, White Plains, NY, 1996; 122–130.
98. Yang C-S, Souter AL, Pollock LL. All-du-path coverage for parallel programs. *International Symposium on Software Testing and Analysis (ISSTA 1998)*, ACM-Software Engineering Notes, Clearwater Beach, Florida, USA, 1998; 153–162.
99. Yang CSD, Pollock LL. All-uses testing of shared memory parallel programs. *Software Testing, Verification and Reliability (STVR)* 2003; **13**(1):3–24.

DOI: 10.1002/cpe

100. Yuan Y, Li ZJ, Sun W. A graph-search based approach to BPEL4WS test generation. *International Conference on Software Engineering Advances (ICSEA 2006)*, Tahiti, French Polynesia, 2006; 14.
101. Humphrey A, Derrick C, Gopalakrishnan G, Tibbitts BR. GEM: Graphical Explorer of MPI Programs. In *5th International Symposium on Software Visualization (SOFTVIS 2010)*. ACM: New York, USA, 2010; 217–218.
102. Edelstein O, Farchi E, Goldin E, Nir Y, Ratsaby G, Ur S. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience* 2003; **15**(3–5):485–499.
103. Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded Java program test generation. *IBM System Journal* 2002; **41**(1):111–125.
104. Wong WE, Lei Y, Ma X. Effective generation of test sequences for structural testing of concurrent programs. *10th IEEE International Conference on Engineering of Complex Systems (ICECCS 2005)*, Shanghai, China, 2005; 539–548.

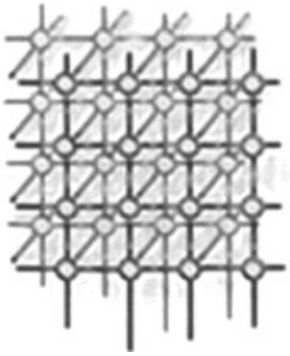
DOI: 10.1002/cpe

Apêndice

C

(Souza et al. 2008b) Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L.; Simao, A. S.; Hausen, A. C. Structural testing criteria for message-passing parallel programs. Concurrency and Computation, 20, p. 1893-1916, 2008b.

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **20**:1893–1916
Published online 17 March 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1297



Structural testing criteria for message-passing parallel programs

S. R. S. Souza[1,*,†], S. R. Vergilio[2], P. S. L. Souza[1], A. S. Simão[1] and A. C. Hausen[2]

[1]*Departamento de Sistemas de Computação, ICMC/USP, São Carlos, SP, Brazil*
[2]*Departamento de Informática, UFPR, Curitiba, PR, Brazil*

SUMMARY

Parallel programs present some features such as concurrency, communication and synchronization that make the test a challenging activity. Because of these characteristics, the direct application of traditional testing is not always possible and adequate testing criteria and tools are necessary. In this paper we investigate the challenges of validating message-passing parallel programs and present a set of specific testing criteria. We introduce a family of structural testing criteria based on a test model. The model captures control and data flow of the message-passing programs, by considering their sequential and parallel aspects. The criteria provide a coverage measure that can be used for evaluating the progress of the testing activity and also provide guidelines for the generation of test data. We also describe a tool, called ValiPar, which supports the application of the proposed testing criteria. Currently, ValiPar is configured for parallel virtual machine (PVM) and message-passing interface (MPI). Results of the application of the proposed criteria to MPI programs are also presented and analyzed. Copyright © 2008 John Wiley & Sons, Ltd.

Received 22 March 2007; Revised 25 November 2007; Accepted 3 December 2007

KEY WORDS: parallel software testing; coverage criteria; testing tool; PVM; MPI

1. INTRODUCTION

Parallel computing is essential to reduce the execution time in many different applications, such as weather forecast, dynamic molecular simulation, bio-informatics and image processing. According to Almasi and Gottlieb [1], there are three basic approaches to build parallel software: (i) automatic

*Correspondence to: S. R. S. Souza, Instituto de Ciências Matemáticas e de Computação, USP Av. Trabalhador São-carlense, 400-Centro Caixa Postal: 668-CEP: 13560-970, São Carlos, SP, Brazil.
†E-mail: srocio@icmc.usp.br

Contract/grant sponsor: CNPq; contract/grant number: 552213/2002-0

environments that generate parallel code from sequential algorithms; (ii) concurrent programming languages such as CSP and ADA; and (iii) extensions for traditional languages, such as C and Fortran, implemented by message-passing environments. These environments include a function library that allows the creation and communication of different processes and, consequently, the development of parallel programs, usually running in a cluster of computers. The most known and used message-passing environments are parallel virtual machine (PVM) [2] and message-passing interface (MPI) [3]. Such environments have gained importance in the last decade and they are the focus of our work.

Parallel software applications are usually more complex than sequential ones and, in many cases, require high reliability levels. Thus, the validation and test of such applications are crucial activities. However, parallel programs present some features that make the testing activity more complex, such as non-determinism, concurrence, synchronization and communication. In addition, the testing teams are usually not trained for testing this class of applications, which makes the test of parallel programs very expensive. For sequential programs, many of the testing problems were reduced with the introduction of testing criteria and the implementation of supporting tools. A testing criterion is a predicate to be satisfied by a set of test cases and can be used as a guideline for the generation of test data. Structural criteria utilize the code, the implementation and structural aspects of the program to select test cases. They are usually based on a control-flow graph (CFG) and definitions and uses of variables in the program [4].

Yang [5] describes some challenges to test parallel programs: (1) developing static analysis; (2) detecting unintentional races and deadlock situations in non-deterministic programs; (3) forcing a path to be executed when non-determinism might exist; (4) reproducing a test execution using the same input data; (5) generating the CFG of non-deterministic programs; (6) providing a testing framework as a theoretical base for applying sequential testing criteria to parallel programs; (7) investigating the applicability of sequential testing criteria to parallel program testing; and (8) defining test coverage criteria based on control and data flows.

There have been some initiatives to define testing criteria for shared memory parallel programs [6–11]. Other works have investigated the detection of race conditions [12–14] and mechanisms to replay testing for non-deterministic programs [15,16]. However, few works are found that investigate the application of the testing coverage criteria and supporting tools in the context of message-passing parallel programs. For these programs, new aspects need to be considered. For instance, data-flow information must consider that an association between one variable definition and one use can occur in different addressing spaces. Because of this different paradigm, the investigation of challenges mentioned above, in the context of message-passing parallel programs, is not a trivial task and presents some difficulties. To overcome these difficulties, we present a family of structural testing criteria for this kind of programs, based on a test model, which includes their main features, such as synchronization, communication, parallelism and concurrency. Testing criteria were defined to exploit the control and data flows of these programs, considering their sequential and parallel aspects. The main contribution of the testing criteria proposed in this paper is to provide a coverage measure that can be used for evaluating the progress of the testing activity. This is important to evaluate the quality of test cases as well as to consider that a program has been tested enough.

The practical application of a testing criterion is possible only if a tool is available. Most existent tools for message-passing parallel programs aid only the simulation, visualization and

DOI: 10.1002/cpe

debugging [16–21]. They do not support the application of testing criteria. To fulfill the demand for tools to support the application of testing criteria in message-passing parallel programming and to evaluate the proposed criteria, we implemented a tool, called ValiPar, which supports the application of the proposed testing criteria and offers two basic functionalities: the selection and evaluation of test data. ValiPar is independent of the message-passing environment and can be configured to different environments and languages. Currently, ValiPar is configured for PVM and MPI programs, in C language. ValiPar was used in an experiment with MPI programs to evaluate the applicability of the proposed criteria, whose results are presented in this paper.

The remainder of this paper is organized as follows. In Section 2, we present the basic concepts and the test model adopted for the definition of the testing criteria. We also introduce the specific criteria for message-passing programs and show an example of usage. In Section 3, the main functionalities of ValiPar are presented and some implementation aspects are discussed. In Section 4, the results of the testing criteria application are presented. In Section 5, related work is presented. Concluding remarks are presented in Section 6.

2. STRUCTURAL TESTING CRITERIA FOR MESSAGE-PASSING PROGRAMS

In this section, we introduce a set of testing criteria defined based on a model that represents the main characteristics of the message-passing parallel programs. This test model is first presented. In order to illustrate the application of the proposed testing criteria, an example of use is presented in Section 2.3.

2.1. Test model and basic concepts

A test model is defined to capture the control, data and communication information of the message-passing parallel programs. This model is based on Yang and Chung's work [11]. The test model considers that a fixed and known number n of processes is created at the initialization of the parallel application. These processes may execute different programs. However, each one executes its own code in its own memory space.

The communication between processes uses two basic mechanisms. The first one is the *point-to-point* communication. A process can send a message to another one using primitives such as *send* and *receive*. The second one is named *collective* communication; a process can send a message to all processes in the application (or to a particular group of them). In our model the collective communication happens in only one pre-defined domain (or context) that includes all the processes in the parallel application. The primitives for collective communication are represented in terms of several basic *sends*.

The parallel program is given by a set of n parallel processes $Prog = \{p^0, p^1, \ldots, p^{n-1}\}$. Each process p has its own control flow graph, CFG^p, which is built by using the same concepts of traditional programs [4]. In short, a CFG of a process p is composed by a set of nodes N^p and a set of edges E^p. These edges that link two nodes of a same process is called intra-process. Each node n in the process p is represented by the notation n^p and corresponds to a set of commands that are sequentially executed or can be associated with a communication primitive (*send*

DOI: 10.1002/cpe

or *receive*). The communication primitives are associated with separate nodes and are represented by the notations `send(p,k,t)` (respectively, `receive(p,k,t)`), meaning that the process p sends (respectively, receives) a message with tag t to (respectively, from) the process k. Note that the model considers blocking and non-blocking receives, such that all possible interleaving between send–receive pairs are represented. The path analysis, described next, permits one to capture the send–receive matching during the parallel program execution.

Each CFG^p has two special nodes: the entry and exit nodes, which correspond to the first and last statements in p, respectively. An edge links a node to another one.

A parallel program *Prog* is associated with a parallel control-flow graph (*PCFG*), which is composed of CFG^p (for $p = 0 \ldots n-1$) and of the representation of the communication between the processes. N and E represent the set of nodes and edges of the *PCFG*, respectively.

Two subsets of N are defined: N_s and N_r, composed of nodes that are associated with *send* and *receive* primitives, respectively. With each $n_i^p \in N_s$, a set R_i^p is associated, such that

$$R_i^p = \{n_j^k \in N_r | \exists \ (\texttt{send(p,k,t)} \text{ at node } n_i^p \text{ and } \texttt{receive(k,p,t)}\, atnode n_j^k),\ \forall k \neq p \wedge k = 0 \ldots n-1\}$$

i.e. R_i^p contains the nodes that can receive a message sent by node n_i^p.

Using the above definitions, we also define the following sets:

- *set of inter-processes edges* (E_s): contains edges that represent the communication between two processes, such that

$$E_s = \{(n_j^{p1}, n_k^{p2}) \mid n_j^{p1} \in N_s, n_k^{p2} \in R_j^{p1}\}$$

- *set of edges* (E): contains all edges, such that

$$E = E_s \cup \bigcup_{p=0}^{n-1} E^p$$

A path π^p in a CFG^p is called an intra-process path. It is given by a finite sequence of nodes, $\pi^p = (n_1^p, n_2^p, \ldots, n_m^p)$, where $(n_i^p, n_{i+1}^p) \in E^p$. $\Pi = (\pi^0, \pi^1, \ldots, \pi^k, S)$ is an inter-processes path of the concurrent execution of *Prog*, where S is the set of synchronization pairs that were executed, such that $S \subseteq E_s$. Observe that the synchronization pairs of S can be used to establish a conceptual path $(n_1^{p1}, n_2^{p1}, \ldots, n_i^{p1}, k_j^{p2} \ldots n_m^{p1})$ or $(k_1^{p2}, k_2^{p^2}, \ldots, n_i^{p1}, k_j^{p2} \ldots k_l^p p2)$. Such paths contain inter-processes edges.

An intra-processes path $\pi^p = (n_1, n_2, \ldots, n_m)$ is simple if all its nodes are distinct, except possibly the first and the last ones. It is loop free if all its nodes are distinct. It is complete if n_1 and n_m are the entry and exit nodes of CFG^p, respectively. We extend these notions to inter-processes paths. An inter-processes path $\Pi = (\pi^0, \pi^1, \ldots, \pi^{n-1}, S)$ is simple if all π^i are simple. It is loop free if all π^i are loop free. It is complete if all π^i are complete. Only complete paths are executed by the test cases, i.e. all the processes execute complete paths. A node, edge or a sub-path is covered (or exercised) if a complete path that includes them is executed.

DOI: 10.1002/cpe

A variable x is defined when a value is stored in the corresponding memory position. Typical definition statements are assignment and input commands. A variable is also defined when it is passed as an output parameter (reference) to a function. In the context of message-passing environments, we need to consider the communication primitives. For instance, the primitive *receive* sets one or more variables with the value t received in the message; thus, this is considered a definition. Therefore, we define:

$$def(n^p) = \{\mathrm{x} \mid \mathrm{x} \text{ is defined in} n^p\}$$

The use of variable x occurs when the value associated with x is referred. The uses can be:

1. a *computational use* (*c-use*): occurs in a computation statement, related to a node n^p in the *PCFG*;
2. a *predicate use* (*p-use*): occurs in a condition (predicate) associated with control-flow statements, related to an intra-processes edge (n^p, m^p) in the *PCFG*; and
3. a *communication use* (*s-use*): occurs in a communication statement (communication primitives), related to an inter-processes edge $(n^{p1}, m^{p2}) \in E_s$.

A path $\pi = (n_1, n_2, \ldots, n_j, n_k)$ is definition clear with respect to (w.r.t.) a variable x from node n_1 to node n_k or edge (n_j, n_k), if $x \in def(n_1)$ and $x \notin def(n_i)$, for $i = 2 \ldots j$.

Similar to traditional testing, we establish pairs composed of definitions and uses of the same variables to be tested [4]. Three kinds of associations are introduced:

c-use association is defined by a triple (n^p, m^p, x), such that $x \in def(n^p)$, m^p has a c-use of x and there is a definition-clear path w.r.t. x from n^p to m^p.

p-use association is defined by a triple $(n^p, (m^p, k^p), x)$, such that $x \in def(n^p)$, (m^p, k^p) has a p-use of x and there is a definition-clear path w.r.t. x from n^p to (m^p, k^p).

s-use association is defined by a triple $(n^{p1}, (m^{p1}, k^{p2}), x)$, such that $x \in def(n^{p1})$, (m^{p1}, k^{p2}) has an s-use of x and there is a definition-clear path w.r.t. x from n^{p1} to (m^{p1}, k^{p2}).

Note that p-use and c-use associations are intra-processes, i.e. the definition and the use of x occur in the same process p. These associations are usually required if we apply the traditional testing criteria to each process separately. An s-use association supposes the existence of a second process and it is an inter-processes association; s-use associations allow the detection of communication faults (in the use of *send* and *receive* primitives). Considering this context, we propose another kind of inter-processes associations to discover communication and synchronization faults:

s–c-use association is given by $(n^{p1}, (m^{p1}, k^{p2}), l^{p2}, x^{p1}, x^{p2})$, where there is an s-use association $(n^{p1}, (m^{p1}, k^{p2}), x^{p1})$ and a c-use association (k^{p2}, l^{p2}, x^{p2}).

s–p-use association is given by $(n^{p1}, (m^{p1}, k^{p2}), (n^{p2}, m^{p2}), x^{p1}, x^{p2})$, where there is an s-use association $(n^{p1}, (m^{p1}, k^{p2}), x^{p1})$ and a p-use association $(k^{p2}, (n^{p2}, m^{p2}), x^{p2})$.

2.2. Structural testing criteria

In this section, we propose two sets of structural testing criteria for message-passing parallel programs, based on test model and definitions presented in previous section. These criteria allow the testing of sequential and parallel aspects of the programs.

DOI: 10.1002/cpe

2.2.1. *Testing criteria based on the control and communication flows*

Each CFG^p (for $p = 0...n-1$) can be tested separately by applying the traditional criteria all-edges and all-nodes. Our objective, however, is also to test the communications in the *PCFG*. Thus, the testing criteria introduced below are based on the types of edges (inter- and intra-processes edges).

- *all-nodes-s criterion*: The test sets must execute paths that cover all the nodes $n_i^p \in Ns$.
- *all-nodes-r criterion*: The test sets must execute paths that cover all the nodes $n_i^p \in Nr$.
- *all-nodes criterion*: The test sets must execute paths that cover all the nodes $n_i^p \in N$.
- *all-edges-s criterion*: The test sets must execute paths that cover all the edges $(n_j^{p1}, n_k^{p2}) \in Es$.
- *all-edges criterion* the test sets must execute paths that cover all the edges $(n_j, n_k) \in E$.

Other criteria could be proposed such as all-paths in the CFG^p and in the *PCFG* (intra- and inter-processes paths). These criteria generally require an infinite number of elements, due to loops in the program. Thus, in such cases, only loop-free paths should be required or selected.

2.2.2. *Testing criteria based on data and message-passing flows*

These criteria require associations between definitions and uses of variables. The objective is to validate the data flow between the processes when a message is passed.

- *all-defs criterion*: For each node n_i^p and each $x \in def(n_i^p)$, the test set must execute a path that covers an association (c-use, p-use or s-use) w.r.t. x.
- *all-defs-s criterion*: For each node n_i^p and each $x \in def(n_i^p)$, the test set must execute a path that covers an inter-processes association (s–c-use or s–p-use) w.r.t. x. In the case where such association does not exist, another one should be selected to exercise the definition of x.
- *all-c-uses criterion*: The test set must execute paths that cover all the c-use associations.
- *all-p-uses criterion*: The test set must execute paths that cover all the p-use associations.
- *all-s-uses criterion*: The test set must execute paths that cover all the s-use associations.
- *all-s–c-uses criterion*: The test set must execute paths that cover all the s–c-use associations.
- *all-s–p-uses criterion*: The test set must execute paths that cover all the s–p-use associations.

Required elements are the minimal information that must be covered to satisfy a testing criterion. For instance, the required elements for the criterion *all-edges-s* are all possible synchronization between parallel processes. However, satisfying a testing criterion is not always possible, due to infeasible elements. An element required by a criterion is infeasible if there is no set of values for the parameters, the input and global variables of the program that executes a path that cover that element. The determination of infeasible paths is an undecidable problem [22].

Non-determinism is another issue that makes the testing activity difficult. An example is presented in Figure 1. Suppose that the nodes 8^1 and 9^1 in p^1 have non-deterministic *receives* and in the nodes 2^0 (p^0) and 2^2 (p^2) have *sends* to p^1. The figure illustrates the possible synchronizations between these processes. These synchronizations represent correct behavior of the application. Therefore, during the testing activity it is essential to guarantee that these synchronizations are executed.

DOI: 10.1002/cpe

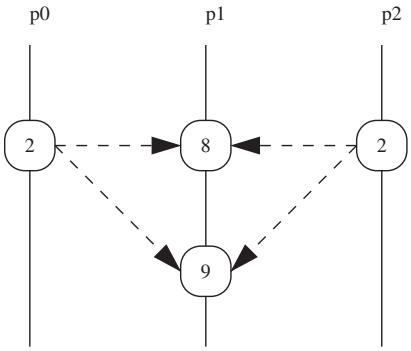


Figure 1. Example of non-determinism.

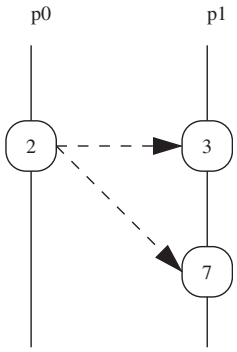


Figure 2. Example of non-blocking receive.

Controlled execution is a mechanism used to achieve deterministic execution, i.e. two executions of the program with the same input are guaranteed to execute the same instruction sequence [15] (and the same synchronization sequence). This mechanism is implemented in ValiPar tool and is described in Section 3.

Figure 2 illustrates an example with *non-blocking receive*. Suppose that the nodes 3^1 and 7^1 in p^1 have *non-blocking receive*. Two synchronization edges are possible, but only one is exercised in each execution. During the path analysis, it is possible to determine the edges that were covered. This information is available in path Π, which is obtained by instrumentation of the parallel program. This instrumentation is described in Section 3.

2.3. An example

In order to illustrate the introduced definitions, consider the GCD program in PVM (Figure 3), described in [23]. This program uses four parallel processes (p^m, p^0, p^1, p^2) to calculate the maximum common divisor of three numbers. The master process p^m (Figure 3(a)) creates the

DOI: 10.1002/cpe

```
/* Master program GCD - mgcd.c */
#include<stdio.h>
#include "pvm3.h"
extern void pack(int);
extern int unpack();
int main(){
/*1*/ int x,y,z, S[3];
/*1*/   scanf("%d%d%d",&x,&y,&z);
/*1*/   pvm_spawn("gcd",(char**)0,0,"",3,S);
/*2*/   pack(&x);
/*2*/   pack(&y);
/*2*/   pvm_send(S[0],1);
/*3*/   pack(&y);
/*3*/   pack(&z);
/*3*/   pvm_send(S[1],1);
/*4*/   pvm_recv(-1,2);
/*4*/   x = unpack();
/*5*/   pvm_recv(-1,2);
/*5*/   y = unpack();
/*6*/   if ((x>1)&&(y>1))        {
/*7*/         pack(&x);
/*7*/         pack(&y);
/*7*/         pvm_send(S[2],1);
/*8*/         pvm_recv(-1,2);
/*8*/         z = unpack();    }
/*9*/   else { pvm_kill(S[2]);
/*9*/          z = 1;    }
/*10*/  printf("%d", z);
/*10*/  pvm_exit();      }
```

(a)

```
/* Slave program GCD - gcd.c */
#include<stdio.h>
#include"pvm3.h"
extern void pack(int);
extern int unpack();
int main(){
/*1*/   int tid,x,y;
/*1*/   tid = pvm_parent();
/*2*/   pvm_recv(tid,-1);
/*2*/   x = unpack();
/*2*/   y = unpack();
/*3*/   while (x != y){
/*4*/     if (x<y)
/*5*/           y = y-x;
/*6*/     else
/*6*/           x = x-y;
/*7*/   }
/*8*/   pack(&x);
/*8*/   pvm_send(tid,2);
/*9*/   pvm_exit();}
```

(b)

Figure 3. GCD program in PVM: (a) master process and (b) slave process.

slave processes p^0, p^1 and p^2, which run 'gcd.c' (Figure 3(b)). Each slave waits (blocked *receive*) two values sent by p^m and calculates the maximum divisor for these values. To finish, the slaves send the calculated values to p^m and terminate their executions. The computation can involve p^0, p^1 and p^2 or only p^0 and p^1, depending on the input values. In p^m, the *receive* commands (nodes 4^m, 5^m and 8^m) are non-deterministic; thus which message will be received in each *receive* command depends on the execution time of each process.

The *PCFG* is presented in Figure 4. The numbers on the left of the source code (Figure 3) represent the nodes in the graph. Inter-processes edges are represented by dotted lines. For simplification reasons, in this figure, only some inter-processes edges (and related *s-use*) are represented. Table I presents the sets $def(n_i^p)$. Table II contains the values of all sets introduced in Section 2.1.

In Table III, we present some elements required by the structural testing criteria introduced in Section 2.2. Test inputs must be generated in order to exercise each possible required element. For example, considering the test input $\{x = 1, y = 2, z = 1\}$, the execution path is $\Pi = (\pi^m, \pi^0, \pi^1, S)$, where $\pi^m = \{1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 9^m, 10^m\}$, $\pi^0 = \{1^0, 2^0, 3^0, 4^0, 5^0, 7^0, 3^0, 8^0, 9^0\}$, $\pi^1 = \{1^1, 2^1, 3^1, 4^1, 6^1, 7^1, 3^1, 8^1, 9^1\}$, $S = \{(2^m, 2^0), (3^m, 2^1), (8^0, 4^m), (8^1, 5^m)\}$. Note that p^2 does not execute any path because the result has been already produced by p^0 and p^1. Owing to the *receive* non-deterministic in nodes 4^m and 5^m, four synchronization edges will be possible: $(8^0, 4^m)$, $(8^0, 5^m)$, $(8^1, 4^m)$, $(8^1, 5^m)$ and only two of them are exercised for each execution of path Π depending on the execution time ($(8^0, 4^m)$ or $(8^0, 5^m)$, $(8^1, 4^m)$ or $(8^1, 5^m)$). In each program execution, it is necessary to determine the inter-processes edges that were executed. This aspect is related to the evaluation of the test cases and was considered in the implementation of ValiPar, described in Section 3.

DOI: 10.1002/cpe

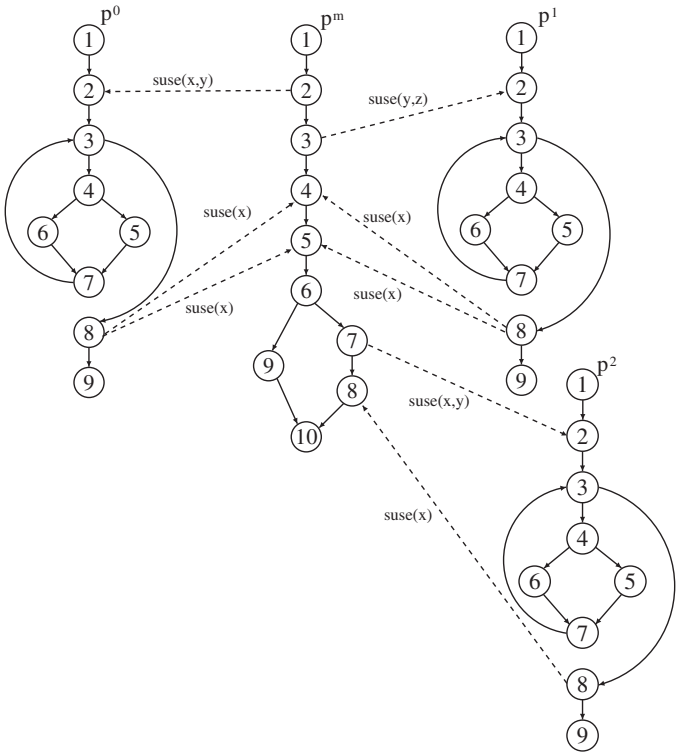


Figure 4. Parallel control-flow graph for GCD program.

Table I. Definition sets for GCD program.

$def(1^m) = \{x, y, z, S\}$	$def(1^0) = \{tid\}$
$def(4^m) = \{x\}$	$def(2^0) = \{x, y\}$
$def(5^m) = \{y\}$	$def(5^0) = \{y\}$
$def(8^m) = \{z\}$	$def(6^0) = \{x\}$
$def(9^m) = \{z\}$	
$def(1^1) = \{tid\}$	$def(1^2) = \{tid\}$
$def(2^1) = \{x, y\}$	$def(2^2) = \{x, y\}$
$def(5^1) = \{y\}$	$def(5^2) = \{y\}$
$def(6^1) = \{x\}$	$def(6^2) = \{x\}$

2.4. Revealing faults

The efficacy (in terms of fault revealing) of the proposed criteria can be illustrated by some kinds of faults that could be present in program GCD (Figure 3) and showing how the criteria contribute to reveal these kinds of faults. The fault situations are based on the works of Howden [24] and

DOI: 10.1002/cpe

Table II. Sets of the test model for GCD program.

$n = 4$
$Prog = \{p^m, p^0, p^1, p^2\}$
$N = \{1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, 10^m, 1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0,$ $8^0, 9^0, 1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2\}$
$N_s = \{2^m, 3^m, 7^m, 8^0, 8^1, 8^2\}$ (nodes with *pvm_send()*)
$N_r = \{4^m, 5^m, 8^m, 2^0, 2^1, 2^2\}$ (nodes with *pvm_recv()*)
$R_2^m = \{2^0, 2^1, 2^2\}$
$R_3^m = \{2^0, 2^1, 2^2\}$
$R_7^m = \{2^0, 2^1, 2^2\}$
$R_8^0 = \{4^m, 5^m, 8^m\}$
$R_8^1 = \{4^m, 5^m, 8^m\}$
$R_8^2 = \{4^m, 5^m, 8^m\}$
$E = E_i^p \cup E_s$
$E_i^m = \{(1^m, 2^m), (2^m, 3^m), (3^m, 4^m), (4^m, 5^m), (5^m, 6^m), (6^m, 7^m),$ $(7^m, 8^m), (8^m, 10^m), (6^m, 9^m), (9^m, 10^m)\}$
$E_i^0 = \{(1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (4^0, 6^0), (5^0, 7^0), (6^0, 7^0),$ $(7^0, 3^0), (3^0, 8^0), (8^0, 9^0)\}$
$E_i^1 = \{(1^1, 2^1), (2^1, 3^1), (3^1, 4^1), (4^1, 5^1), (4^1, 6^1), (5^1, 7^1), (6^1, 7^1),$ $(7^1, 3^1), (3^1, 8^1), (8^1, 9^1)\}$
$E_i^2 = \{(1^2, 2^2), (2^2, 3^2), (3^2, 4^2), (4^2, 5^2), (4^2, 6^2), (5^2, 7^2), (6^2, 7^2),$ $(7^2, 3^2), (3^2, 8^2), (8^2, 9^2)\}$
$E_s = \{(2^m, 2^0), (2^m, 2^1), (2^m, 2^2), (3^m, 2^0), (3^m, 2^1), (3^m, 2^2), (7^m, 2^0),$ $(7^m, 2^1), (7^m, 2^2), (8^0, 4^m), (8^0, 5^m), (8^0, 8^m), (8^1, 4^m), (8^1, 5^m), (8^1, 8^m),$ $(8^2, 4^m), (8^2, 5^m), (8^2, 8^m)\}$

Krawczyk and Wiszniewski [23], which describe typical faults in traditional and parallel programs, respectively.

Howden [24] introduces two types of faults in traditional programs: computation and domain faults. The first one occurs when the result of a computation for an input of the program domain is different from the expected result. The second one occurs when a path that is different from the expected one is executed. For example, in the process slave (gcd.c), replacing the command of node 5^1 '$y = y - x$' by the incorrect command '$y = y + x$' corresponds to a computation fault. A domain fault can be illustrated by changing the predicate $(x < y)$ in edge $(4^1, 5^1)$ by the incorrect predicate $(x > y)$, taking a different path during the execution. These faults are revealed by applying traditional criteria, all-edges, all-nodes, etc., and testing each *CFG* separately. Executing the test input $\{x = 1, y = 2, z = 1\}$ the node 5^1 is covered and the first fault is revealed. Considering the second fault, the test input $\{x = 2, y = 3, z = 2\}$ executes a path that covers the edge $(4^1, 5^1)$ and reveals the fault. For both inputs, the program executes the loop of node 3 (gcd.c) forever, and a failure is produced. These situations illustrate the importance of investigating the application of criteria for sequential testing in parallel software.

In the context of parallel programs, a computation fault can be related to a communication fault. To illustrate this fact, consider that in slave process (Figure 3(b)) the variable y is mistakenly

DOI: 10.1002/cpe

Table III. Some elements required by the proposed testing criteria for GCD program.

all-nodes-*s*	$2^m, 3^m, 7^m, 8^0, 8^1, 8^2$
all-nodes-*r*	$4^m, 5^m, 8^m, 2^0, 2^1, 2^2$
all-nodes	$1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, \ldots, 1^0, 2^0, 3^0, \ldots, 1^1, 2^1, 3^1, \ldots$
all-edges-*s*	$(2^m, 2^0), (2^m, 2^1), (2^m, 2^2), (3^m, 2^0), (3^m, 2^1), (3^m, 2^2), (7^m, 2^2), (8^0, 4^m), (8^0, 5^m),$ $(8^0, 8^m), (8^1, 4^m), (8^1, 5^m), (8^1, 8^m), 8^2, 4^m), (8^2, 5^m), (8^2, 8^m) \ldots$
all-edges	$(1^m, 2^m), (2^m, 3^m), \ldots, (1^0, 2^0), (2^0, 3^0), \ldots, (1^1, 2^1), (2^1, 3^1), \ldots (2^m, 2^0), (2^m, 2^1) \ldots$
all-defs	$(8^m, 10^m, z), (2^0, 5^0, x), (2^0, 6^0, x), (2^0, (3^0, 4^0), x), (2^0, 6^0, y) \ldots$
all-defs-*s*	$(1^m, (2^m, 2^0), 5^0, x, x), (1^m, (2^m, 2^0), 6^0, y, y),$ $(1^m, (2^m, 2^0), (4^0, 5^0), y, y), (1^m, (3^m, 2^0), 5^0, z, y), \ldots$
all-*c*-uses	$(1^m, 10^m, z), (8^m, 10^m, z), (2^0, 8^0, x) \ldots$
all-*p*-uses	$(4^m, (6^m, 7^m), x), (4^m, (6^m, 9^m), x), (5^m, (6^m, 7^m), y),$ $(5^m, (6^m, 9^m), y), (2^0, (3^0, 4^0), x), (2^0, (3^0, 8^0), y) \ldots$
all-*s*-uses	$(1^m, (2^m, 2^0), x, y), (1^m, (2^m, 2^1), x, y), (1^m, (3^m, 2^0), y, z),$ $(4^m, (7^m, 2^2), x), (5^m, (7^m, 2^0), y), (5^m, (7^m, 2^1), y), \ldots$
all-*s*–*c*-uses	$(1^m, (2^m, 2^0), 5^0, x, x), (1^m, (2^m, 2^0), 6^0, x, x), (1^m, (2^m, 2^0), 5^0, y, y),$ $(1^m, (2^m, 2^0), 6^0, y, y), (1^m, (2^m, 2^1), 6^1, x, x),$ $(1^m, (3^m, 2^1), 6^1, x, x), (2^0, (8^0, 8^m), 10^m, x, z), \ldots$
all-*s*–*p*-uses	$(1^m, (2^m, 2^0), (3^0, 4^0), x, x), (1^m, (2^m, 2^0), (3^0, 8^0), x, x),$ $(1^m, (2^m, 2^0), (4^0, 5^0), x, x), (1^m, (3^m, 2^0), (3^0, 4^0), z, y),$ $(5^m, (7^m, 2^0), (3^0, 4^0), y, x), (2^0, (8^0, 4^m), (6^m, 7^m), x, y), \ldots$

replaced by the variable x in communication statement $y = unpack()$ (node 5^m). The received value is written in the same variable received previously (variable x). Some test inputs, such as $\{x = 1, y = 2, z = 1)\}$, do not reveal this fault. However, this fault can be revealed when we apply, for example, the all-defs-*s* criterion. The test input $\{x = 2, y = 8, z = 4\}$, which covers the association $(5^m, (7^m, 2^2), 5^2, y, y)$, reveals this fault.

Krawczyk and Wiszniewski [23] present two kinds of faults related to parallel programs: observability and locking faults. The observability fault is a special kind of domain fault, related to synchronization faults. These faults can be observed or not during the execution of a same test input; the observation depends on the parallel environment and on the execution time (non-determinism). Locking faults occur when the parallel program does not finish its execution, staying locked, waiting forever. To illustrate this fault, consider again the execution of the program GCD with the test input $\{x = 7, y = 14, z = 28\}$. The expected output is (7) and the expected matching points between *send–receive* pairs are $(2^m, 2^0), (3^m, 2^1), (8^0, 4^m)$ or $(8^0, 5^m), (8^1, 5^m)$ or $(8^1, 4^m), (7^m, 2^2), (8^2, 8^m)$. It is important to point out that nodes 4^m and 5^m have non-deterministic *receive* primitives (Section 2.3).

Without loss of generality, let us consider that the matching points reached are $(8^0, 4^m)$ and $(8^1, 5^m)$. Suppose that in node 5^m the statement *pvm_recv*() has been mistakenly changed to *pvm_nrecv*(), a non-blocking primitive. In this case, the message sent by slave p^1 may be not reached by non-blocking *receive* in node 5^m, before the execution of this node. This is a synchronization fault. Thus, variable y is not updated with the value sent from slave p^1. This fact could appear irrelevant here, since the value of y (14) is equal to the value that must be received

DOI: 10.1002/cpe

from p^1. However, this fault makes the node 8^m to receive the message from 8^1 instead of the message from 8^2. This fault can be revealed by the all-s-uses criterion. To cover the s-use association $(6^2, (8^2, 8^m), x)$, the tester has to provide a test input that executes the slave process p^2, for instance, $\{x = 3, y = 9, z = 4\}$. The expected output (1) is obtained, but the s-use association is not covered (due to the fault related to the non-blocking *receive*). This test case did not reveal the fault, but it indicated an unexpected path. The tester must try to select a test input that covers the s-use association. The test input $\{x = 7, y = 14, z = 28\}$ covers the association and also produces an unexpected output. The tester can conclude that the program has a fault. ValiPar (discussed in Section 3) provides support in this case, allowing the analysis of the execution trace. By analyzing the execution trace, the tester can observe that a wrong matching point was reached.

This fault is related to non-determinism and the occurrence of the illustrated matching points is not guaranteed. For example, if the slave process p^1 is fast enough to execute, the sent message reaches the node 5^m and the fault will not be observed. Notwithstanding, the synchronizations illustrated previously are more probable, considering the order of the processes creation.

A special type of the locking error is deadlock [25], a classical problem in parallel programs. Ideally, it must be detected before the parallel program execution. It is not the focus of the testing criteria proposed in this work; nonetheless, the information extracted from the parallel programs during the application of the coverage criteria may be used to statically detect deadlock situations.

3. ValiPar TESTING TOOL

To support the effective application of the testing criteria defined in the previous section, we have implemented ValiPar. ValiPar works with the concept of test sessions, which can be set up to test a given parallel program and allows one to stop testing activity and resume it later. Basically, the tool provides functionalities to (i) create test sessions, (ii) save and execute test data and (iii) evaluate the testing coverage w.r.t. a given testing criterion.

The implementation of the tool follows the architecture shown in Figure 5. This architecture was also described in [26]. ValiPar has four main modules: *ValiInst* performs all static analysis of parallel program; *ValiElem* generates the list of required elements; *ValiEval* performs test case evaluation (coverage computation); and *ValiExec* involves the parallel program execution (virtual machine creation) and generation of the executed paths.

ValiPar is able to validate parallel programs in different message-passing environments with a fixed number of processes. It is currently instanced for PVM and MPI parallel programs in C language. To adapt this tool for another message-passing environment or programming language, it is required to instance the modules *ValiInst* and *ValiExec*.

3.1. ValiInst

The ValiInst module is responsible for extracting flow information of the parallel program and for instrumenting the program with statements that will register the actual paths of execution. These tasks are accomplished mostly using the `idelgen` system, which is a compiler for the IDeL language (*Instrumentation Description Language*) [27]. IDeL is a meta-language that can

DOI: 10.1002/cpe

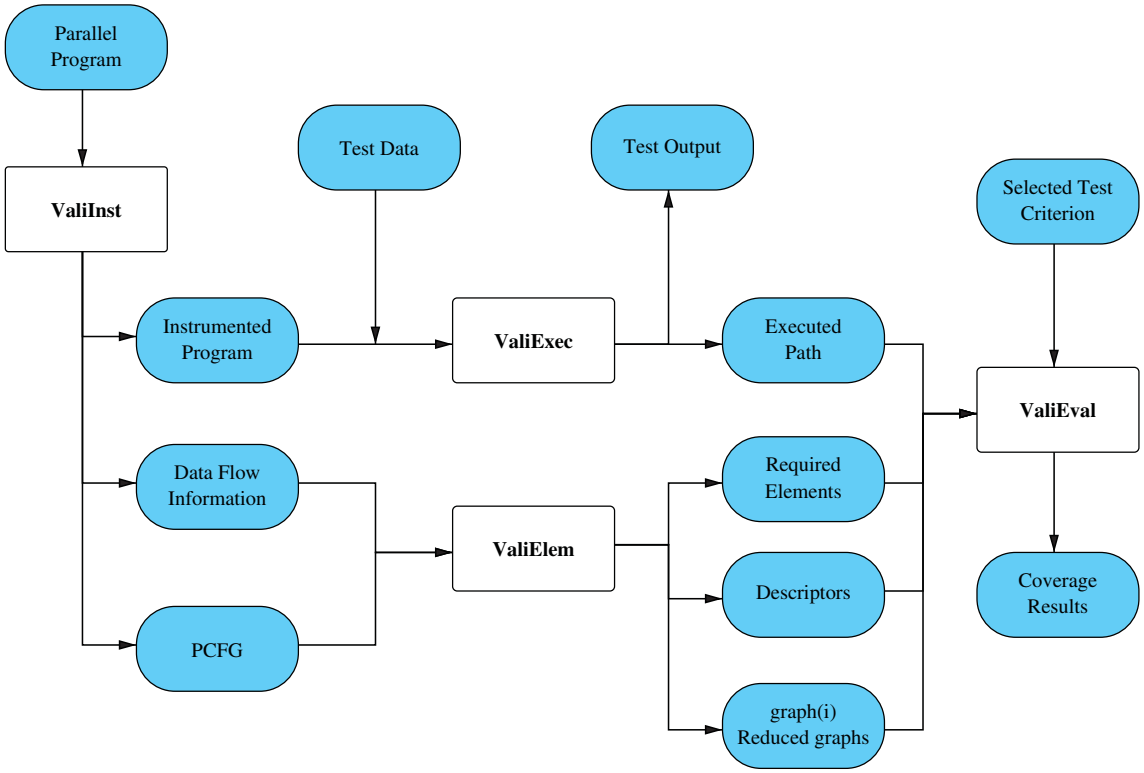


Figure 5. ValiPar tool architecture.

be instanced for different languages. In the context of this work, the instantiation of IDeL for C language was used and it was extended to treat specific aspects of PVM and MPI.

The *PCFG* is generated with information about nodes, edges, definitions and uses of variables in the nodes, as well as the presence of *send* and *receive* primitives[‡]. In this version of ValiPar the primitives for collective communication were not implemented. They need to be mapped in terms of *send* and *receive* basics.

This information set is generated for each process. The `idelgen` accomplishes the syntactic and semantic analysis of the program, according to the grammar of a given language, extracting the necessary information for instrumentation. The instrumented program is obtained by inserting check-point statements in the program being tested. These statements do not change the program semantics. They only write necessary information in a trace file, by registering the node and the process identifier in the *send* and *receive* commands. The instrumented program will produce the paths executed in each process, as well as the synchronization sequence produced within a test case.

[‡]The following primitives were considered. For MPI: *MPI_send()*, *MPI_Isend()*, *MPI_recv()* and *MPI_Irecv()*; for PVM: *pvm_send()*, *pvm_recv()* and *pvm_nrecv()*.

DOI: 10.1002/cpe

3.2. ValiElem

The ValiElem module generates the required elements by the coverage testing criteria defined in this paper. These elements are generated from *PCFGs* and data-flow information, generated by ValiInst. For that purpose, two other graphs are used: the heirs reduced graph, proposed by Chusho [28], and the graph(i), used by the testing tool Poketool [29].

In a reduced graph of heirs, all the branches are primitive. The algorithm is based on the fact that there are edges inside a *PCFG*, which are always executed when another one is executed. If each complete path that includes the edge a also includes the edge b, then b is called heir of a, and a is called ancestral of b, because b inherits information about execution of a. In other words, an edge that is always executed when another one is executed is called heir edge. An edge is called primitive, if it is not heir of any other one. ValiPar adapted the algorithm for the parallel programs context. The concept of synchronization edge was included to the concept of primitive edge. Minimizing the number of edges required by ValiPar is possible by the use of both concepts.

A graph(i) is built for each node that contains a variable definition. The purpose of this is to obtain all definition-clear paths w.r.t. a variable $x \in def(n_i^p)$. Hence, a given node k will belong to a graph(i) if at least one path from i to k exists and this path does not redefine at least one variable `x`, defined in i. A node k can generate several different images in the graph because just one graph(i) is built for all defined variables in node i. However, the paths in the graph(i) are simple. To do this and to avoid infinite paths, caused by the existence of loops in the *CFG*, in the same path of the graph(i) only a node can contain more than one image, and its image is the last node of the path. The graph(i) is used to establish associations between definitions and uses of variables, generating the elements required by the data-flow testing criteria introduced in Section 2.

For each required element, ValiElem also produces a descriptor, which is a regular expression that describes a path that exercises a required element. For instance, the descriptor for the elements required by all-nodes criterion is given by the expression:

$$N * n_i^p N *$$

where N is the set of nodes in CFG^p. A required node n_i^p will be exercised by the path π^p, if π^p includes n_i. In the same way, a regular expression is defined for each element required by all testing criteria.

The descriptor describes all the paths in the graph that exercise the corresponding element and is used by ValiEval module. Figure 6 shows the required elements generated for the all-edges-s criterion, considering the program in Figure 3

Note that, in this section, we follow the notation that is adopted in the tool. For instance, `2–0` means node 2 in process 0. Moreover, the master process is always represented by process 0 and the slave processes are appropriately named 1, 2, 3, ... and so on.

3.3. ValiExec

ValiExec executes the instrumented program with the test data provided by the user. A script is used to initialize the message-passing environment before parallel program execution. ValiExec stores the test case, the execution parameters and the respective execution trace. The execution

DOI: 10.1002/cpe

1) 2–0 2–1	7) 7–0 2–1	13) 8–1 2–2	19) 8–2 2–3
2) 2–0 2–2	8) 7–0 2–2	14) 8–1 2–3	20) 8–3 4–0
3) 2–0 2–3	9) 7–0 2–3	15) 8–2 4–0	21) 8–3 5–0
4) 3–0 2–1	10) 8–1 4–0	16) 8–2 5–0	22) 8–3 8–0
5) 3–0 2–2	11) 8–1 5–0	17) 8–2 8–0	23) 8–3 2–1
6) 3–0 2–3	12) 8–1 8–0	18) 8–2 2–1	24) 8–3 2–2

Figure 6. Required elements of all-edges-s criterion.

```
traceP0 :
1–0   2–0   3–0   4–0   8–1   4–0   5–0   8–2   5–0   6–0   9–0
   10–0

traceP1 :
1–1   2–1   2–0   2–1   3–1   4–1   5–1   3–1   4–1   5–1   3–1
   7–1   8–1   9–1

traceP2 :
1–2   2–2   3–0   2–2   3–2   4–2   5–2   3–2   4–2   6–2   3–2
   4–2   5–2   3–2   7–2   8–2   9–2

traceP3 :
```

Figure 7. Trace file.

trace includes the executed path of each parallel process, as well as the synchronization sequences. It will be used by ValiEval to determine the elements that were covered.

After the execution, the tester can visualize the outputs and the execution trace to determine whether the obtained output is the same as that expected. If it is not, a fault was identified and may be corrected before continuing the test.

A trace of a parallel process is represented by a sequence of nodes executed in this process. A synchronization from n_i^a to m_j^b is represented at the trace of the sender process of the message by the sequence $n_{i-1}^a\ n_i^a\ m_j^b\ n_i^a\ n_{i+1}^a$. Note that process a is unable to know to which node j of process b the message was sent. The same synchronization is represented at the trace of the receiver process by the sequence $m_{j-1}^b\ m_j^b\ n_i^a\ m_j^b\ m_{j+1}^b$. In this way, it is possible to determine whether the inter-processes edge (n_i^a, m_j^b) was covered. The produced traces are used to evaluate the test cases and they provide a way for debugging the program. To illustrate, Figure 7 shows the traces generated for GCD program, executed with the test input: $\{x = 1, y = 3, z = 5\}$. For this test, process 3 was not executed.

ValiExec also enables the controlled execution of the parallel program under test. This feature is useful for replaying the test activity. Controlled execution guarantees that two executions of the parallel program with the same input will produce the same paths and the same synchronization sequences. The implementation of controlled execution is based on the work of Carver and Tai [15], adapted to message-passing programs. Synchronization sequences of each process are gathered in runtime by the instrumented check-points of blocking and non-blocking *sends* and *receives*. The latter is also subject to non-determinism; hence, each request is associated with the number of times it has been evaluated. This information and other program inputs are used to achieve the deterministic execution and, thus, to allow test case replay.

DOI: 10.1002/cpe

3.4. ValiEval

ValiEval evaluates the coverage obtained by a test case set w.r.t. a given criterion. ValiEval uses the descriptors, the required elements generated by ValiElem and the paths executed by the test cases to verify which elements required for a given testing criterion are exercised. The module implements the automata associated with the descriptors. Thus, a required element is covered if an executed path is recognized by its corresponding automaton. The coverage score (percentage of covered elements) and the list of covered and not covered elements for the selected test criterion is provided as output. Figure 8 shows this information considering the all-edges-*s* criterion and the GCD program (Figure 3). These results were generated after the execution of test inputs in Figure 9.

```
Required elements not covered - criterion all_edges_s:
Required element 2 not covered: 2) 2-0 2-2
Required element 3 not covered: 3) 2-0 2-3
Required element 4 not covered: 4) 3-0 2-1
Required element 6 not covered: 6) 3-0 2-3
Required element 8 not covered: 8) 7-0 2-2
Required element 9 not covered: 9) 7-0 2-3
Required element 11 not covered: 11) 8-1 4-0
Required element 13 not covered: 13) 8-1 2-2
Required element 14 not covered: 14) 8-1 2-3
Required element 15 not covered: 15) 8-2 4-0
Required element 17 not covered: 17) 8-2 8-0
Required element 18 not covered: 18) 8-2 2-1
Required element 19 not covered: 19) 8-2 2-3
Required element 20 not covered: 20) 8-3 5-0
Required element 21 not covered: 21) 8-3 8-0
Required element 23 not covered: 23) 8-3 2-1
Required element 24 not covered: 24) 8-3 2-2

Coverage: 29.17%
```

Figure 8. Informations about coverage of the all-edges-s criterion.

```
input1.tes:
1 3 5
output1.tes:
1
input2.tes:
2 8 4
output2.tes:
2
input3.tes:
5 1 2
output3.tes:
1
input4.tes:
4 4 4
output3.tes:
4
```

Figure 9. Test cases executed for GCD program.

DOI: 10.1002/cpe

3.5. Testing procedures with ValiPar

ValiPar tool and proposed criteria can be applied following two basic procedures: (1) to guide the selection of test cases to the program and (2) to evaluate the test set quality, in terms of code and communication coverage.

1. *Test data selection with ValiPar*: Suppose that the tester uses ValiPar for supporting the test data selection. For this, the following steps must be conducted:

(a) Choose a testing criterion to guide the test data selection.
(b) Identify test data that exercise the elements required by the testing criterion.
(c) For each test case, analyze if the output is correct; otherwise, the program must be corrected.
(d) While uncovered required elements exist, identify new test cases that exercise each one of them.
(e) The tester proceeds with this method until the desired coverage is obtained (ideally 100%). In addition, other testing criteria may be selected to improve the quality of the generated test cases.

In some cases, the existence of infeasible elements does not allow a 100% coverage of a criterion. The determination of infeasible elements is an undecidable problem [22]. Because of this, the tester has to manually determine the infeasibility of the paths and required elements.

2. *Test data evaluation with ValiPar*: Suppose that the tester has a test set T and wishes to know how good it is, considering a particular testing criterion. Another possible scenario is that the tester wishes to compare two test sets T_1 and T_2. The coverage w.r.t. a testing criterion can be used in both cases. The tester can use ValiPar in the following way:

(a) Execute the program with all test cases of T (or T_1 and T_2) to generate the execution traces or executed paths.
(b) Select a testing criterion and evaluate the coverage of T (or the coverage of T_1 and T_2).
(c) If the coverage obtained is not the expected, the tester can improve this coverage by generating new test data.
(d) To compare sets T_1 and T_2, the tester can proceed as before, creating a test session for each test set and then comparing the coverage obtained. The greater the coverage obtained, the better the test set.

Note that these procedures are not exclusive. If an *ad hoc* test set is available, it can be evaluated according to Procedure 2. If the obtained coverage is not adequate, this set can be improved by using Procedure 1. The use of such an initial test set allows effort reduction in the application of the criteria. In this way, our criteria can be considered complementary to *ad hoc* approaches. They can improve the efficacy of the test cases generated by *ad hoc* strategies and offer a coverage measure to evaluate them. This measure can be used to know whether a program has been tested enough and to stop testing.

4. APPLICATION OF TESTING CRITERIA

In this section, we present the results of the application of the criteria for message-passing parallel programs. The objective is to evaluate the proposed criteria costs in terms of the test set sizes and

DOI: 10.1002/cpe

number of required elements. Although this issue would need a broader range of studies to achieve statistically significant results, the current work provides evidences of the applicability of the testing criteria proposed herein.

Five programs implemented in MPI were used: (1) *gcd*, which calculates the greatest common divisor of three numbers (example used in Figure 4); (2) *phil*, which implements the dining philosophers problem (five philosophers); (3) *prod-cons*, which implements a multiple-producer single-consumer problem; (4) *matrix*, which implements multiplication of matrix; (5) *jacobi*, which implements the iterative method of the Gauss–Jacobi for solving a linear system of equations. These programs represent concurrent-programming classical problems. Table IV shows the complexity of the programs, in terms of the number of parallel processes and the number of *receive* and *send* commands.

For each program, an initial test set (T_i) was randomly generated. Then, T_i was submitted to ValiPar (version MPI) and an initial coverage was obtained for all the criteria. After this, additional test cases (T_a) were generated to cover the elements required by each criterion and not covered by T_i. The final coverage was then obtained. In this step, the infeasible elements were detected with support of the controlled execution. Table V presents the number of covered and infeasible elements for the testing criteria. The adequate set was obtained from $T_i \cup T_a$ by taking only the test cases that really contributed to cover elements in the executed order. The size of the adequate sets is presented in Table VI.

Table IV. Characteristics of the case studies.

Programs	Processes	Sends	Receives
gcd	4	7	7
phil	6	36	11
prod-cons	4	3	2
matrix	4	36	36
jacobi	4	23	31

Table V. Number of covered and infeasible elements for the case studies.

	Covered elements/infeasible elements				
Testing criteria	gcd	phil	prod-cons	matrix	jacobi
all-nodes	62/0	176/0	60/0	368/200	499/19
all-nodes-*r*	7/0	11/0	2/0	36/15	31/2
all-nodes-*s*	7/0	36/0	3/0	36/21	23/2
all-edges	41/20	356/280	21/0	1032/982	652/499
all-edges-*s*	30/20	325/280	6/0	972/945	531/492
all-*c*-uses	29/0	50/0	43/2	572/337	608/77
all-*p*-uses	40/0	148/27	42/2	304/206	514/118
all-*s*-uses	66/47	335/280	6/0	1404/1375	768/729

DOI: 10.1002/cpe

Table VI. Size of effective test case sets.

	Size of adequate test sets				
Testing criteria	gcd	phil	prod-cons	matrix	jacobi
all-nodes	6	2	2	2	7
all-nodes-*r*	2	1	2	1	3
all-nodes-*s*	2	2	1	1	3
all-edges	3	2	2	2	7
all-edges-*s*	3	2	2	1	3
all-*c*-uses	6	2	2	2	9
all-*p*-uses	9	4	3	2	9
all-*s*-uses	10	2	2	3	6

By analyzing the results, we observe that the criteria are applicable. In spite of the great number of required elements for the programs *phil*, *matrix* and *jacobi*, the number of test cases does not grow proportionally. The size of the adequate test sets is small.

In fact, some effort is necessary to identify infeasible elements. In this study, the controlled execution was used to aid in the identification of the infeasible elements. A good strategy is to analyze the required elements to decide infeasibility only when the addition of new test cases does not contribute to improve coverage. In this case, paths are identified to cover the remaining elements and, if possible, specific test cases are generated. Other strategy is to use infeasible patterns for classification of the paths. Infeasible patterns are structures composed of sequence of nodes with inconsistent conditions [30]. The use of patterns is an important mechanism to identify infeasibility in traditional programs. If a path contains such patterns it will be infeasible. In order to reduce the problem of infeasible paths, we intend to implement in ValiPar a mechanism for automatically discarding infeasible paths according to a pattern provided by the tester.

We observed, in the results of the experiment, that many infeasible elements are related to the *s*-uses (*all-edges-s* and *all-s-uses* criteria). This situation occurs because we adopted a conservative position by generating all the possible inter-processes edges, even when the communication may not be possible in the practice. This was adopted with the objective of revealing faults related to missing communications. We are now implementing a mechanism to disable the generation of all the combinations, if desired by the tester. Another idea is to generate all possible communication uses (*s*-uses) during the static analysis and, during the program execution, to obtain which *s*-uses tried to synchronize (race situation). These *s*-uses that participate in the race have high probability of being feasible; otherwise, *s*-uses have major probability of being infeasible. This investigation is inspired on the work of Damodaran-Kamal and Francioni [16].

5. RELATED WORK

Motivated by the fact that traditional testing techniques are not adequate for testing features of concurrent/parallel programming, such as non-determinism and concurrency, many researchers have developed specific testing techniques addressing these issues.

DOI: 10.1002/cpe

Lei and Carver [14] present a method that guarantees that every partially ordered synchronization will be exercised exactly once without saving any sequences that have already been exercised. The method is based on the reachability testing. By definition, the approach avoids generation of unreachable testing requirements. Their method is complementary to our approach. On the one hand, the authors employ a reachability schema to calculate the synchronization sequence automatically. They do not address how to select the test case which will be used for the first run. On the other hand, we use the static analysis of the program to indicate the test cases that are worth selecting. Therefore, the coverage metrics we proposed can be used to derive the test case suite that will be input to the reachability-based testing, as argued by the authors.

Wong *et al.* [31] propose a set of methods to generate test sequences for structural testing of concurrent programs. The reachability graph is used to represent the concurrent program and to select test sequences to the all-node and all-edge criteria. The methods aim the generation of a small test sequences set that covers all the nodes and the edges in a reachability graph. For this, the methods provide information about which parts of the program should be covered first to effectively increase the coverage of these criteria. The authors stress that the major advantage of the reachability graph is that only feasible paths are generated. However, the authors do not explain how to generate the reachability graph from the concurrent program or how to deal with the state space explosion.

Yang and Chung [11] introduce the path analysis testing of concurrent programs. Given a program, two models are proposed: (1) *task flow graph*, which corresponds to the syntactical view of the task execution behavior and models the task control flow, and (2) *rendezvous graph*, which corresponds to the runtime view and models the possible rendezvous sequences among tasks. An execution of the program will traverse one concurrent path of the rendezvous graph (*C-route*) and one concurrent path of the flow graph (*C-path*). A method called *controlled execution* to support the debugging activity of concurrent programs is presented. They pointed out three research issues to be addressed to make their approach practical: *C*-path selection, test generation and test execution.

Taylor *et al.* [8] propose a set of structural coverage criteria for concurrent programs based on the notion of concurrent states and on the concurrency graph. Five criteria are defined: *all-concurrency-paths*, *all-proper-cc-histories*, *all-edges-between-cc-states*, *all-cc-states* and *all-possible-rendezvous*. The hierarchy (subsumption relation) among these criteria is analyzed. They stress that every approach based on reachability analysis would be limited in practice by state space explosion. They mentioned some alternatives to overcome the associated constraints.

In the same vein of Taylor and colleagues' work, Chung *et al.* [6] propose four testing criteria for Ada programs: *all-entry-call*, *all-possible-entry-acceptance*, *all-entry-call-permutation* and *all-entry-call-dependency-permutation*. These criteria focus the rendezvous among tasks. They also present the hierarchy among these criteria.

Edelstein *et al.* [12,13] present a multi-threaded bug detection architecture called *ConTest* for Java programs. This architecture combines a replay algorithm with a seeding technique, where the coverage is specific to race conditions. The seeding technique seeds the program with *sleep* statements at shared memory access and synchronization events and heuristics are used to decide when a *sleep* statement must be activated. The replay algorithm is used to re-execute a test when race conditions are detected, ensuring that all accesses in race will be executed. The focus of the work is the non-determinism problem, not dealing with code coverage and testing criteria.

DOI: 10.1002/cpe

Yang *et al.* [9,10] extend the data-flow criteria [4] to shared memory parallel programs. The parallel program model used consists of multiple threads of control that can be executed simultaneously. A *parallel program-flow graph* is constructed and is traversed to obtain the paths, variable definitions and uses. All paths that have definition and use of variables related with parallelism of threads constitute test requirements to be exercised. The *Della Pasta Tool* (*Delaware Parallel Software Testing Aid*) automates their approach. The authors presented the foundations and theoretical results for structural testing of parallel programs, with definition of the all-du-path and all-uses criteria for shared memory programs. This work inspired the test model definition for message-passing parallel programs, described in Section 2.

The previous works stress the relevance of providing coverage measures for concurrent and parallel programs, considering essentially shared memory parallel programs. They do not address coverage criteria that consider the main features of the message-passing programs. Our work is based on the works mentioned above, but differently we explore control and data-flow concepts to introduce criteria specific for the message-passing environment paradigm and describe a supporting tool.

A related, but orthogonal, approach to testing is the use of model checking methods to provide evidences of the correctness of an algorithm, by suitably exploring the state space of all possible executions [32]. Improvements in model checking theory and algorithms allow handling huge state space. When effectively done, model checking can provide a slightly stronger assertion on the correctness of parallel programs than testing with selected test cases. There exist some initiatives of model checking of parallel programs [33–36]. These approaches suffer from several drawbacks, though. Firstly, the program cannot usually be model-checked directly, requiring instead the conversion into a suitable model. This conversion is rarely automated and must be made manually [36]. However, in this case, it is the correction of the model that is analyzed, not of the actual program. It remains to be demonstrated that the model correctly represents the program. Sometimes, the model is difficult to obtain, since important primitives of parallel program may not be directly represented in the model. This problem has been recently tackled in [34], where an extension to the model checker SPIN, called MPI-SPIN, is proposed. Although the gap between the program and the model is reduced, a direct translation is far from being feasible. Another drawback of model checking is the awkward handling of user inputs. There exist some approaches that use symbolic execution in order to represent all possible user inputs symbolically, e.g. [33]. Nonetheless, symbolic execution is a long-term research topic and brings its own problems, since the expression obtained along the paths grows intractable. Then, even if model checking is used in the verification of some model of the program, the testing of the program is still important, and the problem of measuring the quality of the test cases used to test the program still remains.

In relation to parallel testing tools, most tools available aid only the simulation, visualization and debugging; they do not support the application of testing criteria. Examples of these tools are TDC Ada [20] and ConAn [19], respectively, for ADA and Java. For message-passing environments, we can mention Xab [17], Visit [18] and MDB [16] for PVM, and XMPI [37] and Umpire [21] for MPI.

When we consider testing criteria support, we can mention the tool Della Pasta [9], based on threads, and the tool STEPS [38]. This last one works with PVM programs and generates paths to cover some elements in the control-flow graphs of PVM programs. We could not find in the

DOI: 10.1002/cpe

Table VII. Existent testing tools.

Tool	Data flow	Control flow	Test replay	Debug	Language
TDC Ada			√		Ada
ConAn			√		Java
Della Pasta	√		√	√	C
Xab				√	PVM
Visit				√	PVM
MDB			√	√	PVM
STEPS		√	√	√	PVM
Astral		√		√	PVM
XMPI				√	MPI
Umpire				√	MPI
ValiPar	√	√	√	√	PVM and MPI

literature a tool, which implements criteria based on control, data and communication flows, as the one presented in this paper. Table VII shows the main facilities of ValiPar, compared with the existing tools.

6. CONCLUDING REMARKS

Testing parallel programs is not a trivial task. As mentioned previously, to perform this activity some problems need to be investigated. This paper contributes in this direction by addressing some of them in the context of message-passing programs: definition of a model to capture relevant control and data-flow information and to statically generate the corresponding graph; proposition of specific testing coverage criteria; development of a tool to support the proposed criteria, as well as, sequential testing; implementation of mechanisms to reproduce a test execution and to force the execution of a given path in the presence of non-determinism; and evaluation of the criteria and investigation of the applicability of the criteria.

The proposed testing criteria are based on models of control and data flows and include the main features of the most used message-passing environments. The model considers communication, concurrency and synchronization faults between parallel processes and also fault related to sequential aspects of each process.

The use of the proposed criteria contributes to improve the quality of the test cases. The criteria offer a coverage measure that can be used in two testing procedures. The first one for the generation of test cases, where these criteria can be used as guideline for test data selection. The second one is related to the evaluation of a test set. The criteria can be used to determine when the testing activity can be ended and also to compare test sets. This work also showed that the testing criteria can contribute to reveal important faults related with parallel programs.

The paper described ValiPar, a tool that supports the proposed criteria. ValiPar is independent of the message-passing environment and is currently configured for PVM (ValiPVM) and MPI (ValiMPI). These versions are configured for language C. We intend to configure other versions of ValiPar, considering others languages used for message-passing parallel programs, e.g. Fortran.

DOI: 10.1002/cpe

Non-determinism is very common in parallel programs and causes problems for validation activity. To minimize these problems, we implemented in ValiPar mechanisms to permit controlled execution of parallel programs. With these mechanisms, synchronization sequences can be re-executed, repeating the test and, thus, contributing for the revalidation and regression testing of the parallel programs.

Using the MPI version of ValiPar, we carried out a case study that showed the applicability of the proposed criteria. The results showed a great number of required elements mainly for the communication-flow-based criteria. This should be evaluated in future experiments and some refinements may be proposed to the criteria. We intend to conduct other experiments to explore efficacy aspects to propose changes in the way of generating the required elements and to avoid a large number of infeasible ones.

The advantage of our coverage criteria, comparing with another techniques for testing parallel programs, is to systematize the testing activity. In fact, there exists an amount of cost and time associated with the application of the coverage criteria. However, the criteria provide a coverage measure that can be used to assess the quality of the tests conducted. In the case of critical applications, this evaluation is fundamental. In addition, ValiPar reduces this cost, by automating most of the activities related on parallel program testing.

The evolution of our work on this subject is directed to several lines of research: (1) development of experiments to refine and evaluate the testing criteria; (2) use of ValiPar for real and more complex parallel programs; (3) implementation of mechanisms to validate parallel programs that dynamically create processes and other ones to help the tester in identifying infeasible elements; (4) conduction of an experiment to evaluate the efficacy of the generated test data against *ad hoc* test sets; and (5) definition of a strategy that synergistically combines model checking methods and the testing criteria.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their helpful comments and Felipe Santos Sarmanho for his assistance in the experiments. This work was supported by Brazilian funding agency CNPq, under Process number 552213/2002-0.

REFERENCES

1. Almasi GS, Gottlieb A. *Highly Parallel Computing* (2nd edn). The Benjamin Cummings Publishing Company: Menlo Park, CA, 1994.
2. Geist GA, Kohl JA, Papadopoulos PM, Scott SL. Beyond PVM 3.4: What we've learned what's next, and why. *Fourth European PVM–MPI Conference—Euro PVM/MPI'97*, Cracow, Poland, 1997; 116–126.
3. Snir M, Otto S, Steven H, Walker D, Dongarra J. MPI: The complete reference. *Technical Report*, MIT Press, Cambridge, MA, 1996.
4. Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* 1985; **11**(4):367–375.
5. Yang C-SD. Program-based, structural testing of shared memory parallel programs. *PhD Thesis*, University of Delaware, 1999.
6. Chung C-M, Shih TK, Wang Y-H, Lin W-C, Kou Y-F. Task decomposition testing and metrics for concurrent programs. *Fifth International Symposium on Software Reliability Engineering* (*ISSRE'96*), 1996; 122–130.
7. Koppol PV, Carver RH, Tai K-C. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering* 2002; **28**(6):607–623.

8. Taylor RN, Levine DL, Kelly C. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering* 1992; **18**(3):206–215.
9. Yang C-S, Souter AL, Pollock LL. All-du-path coverage for parallel programs. *International Symposium on Software Testing and Analysis* (*ISSTA'98*), *ACM-Software Engineering Notes*, 1998; 153–162.
10. Yang C-SD, Pollock LL. All-uses testing of shared memory parallel programs. *Software Testing*, *Verification and Reliability* (*STVR*) 2003; **13**(1):3–24.
11. Yang R-D, Chung C-G. Path analysis testing of concurrent programs. *Information and Software Technology* 1992; **34**(1).
12. Edelstein O, Farchi E, Goldin E, Nir Y, Ratsaby G, Ur S. Framework for testing multi-threaded java programs. *Concurrency and Computation*: *Practice and Experience* 2003; **15**(3–5):485–499.
13. Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded java program test generation. *IBM System Journal* 2002; **41**(1):111–125.
14. Lei Y, Carver RH. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering* 2006; **32**(6):382–403.
15. Carver RH, Tai K-C. Replay sand testing for concurrent programs. *IEEE Software* 1991; 66–74.
16. Damodaran-Kamal SK, Francioni JM. Nondeterminacy: Testing and debugging in message passing parallel programs. *Third ACM/ONR Workshop on Parallel and Distributed Debugging*. ACM Press: New York, 1993; 118–128.
17. Beguelin AL. XAB: A tool for monitoring PVM programs. *Workshop on Heterogeneous Processing—WHP03*. IEEE Press: New York, April 1993; 92–97.
18. Ilmberger H, Thürmel S, Wiedemann CP. Visit: A visualization and control environment for parallel program debugging. *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993; 199–201.
19. Long B, Hoffman D, Strooper P. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering* 2003; **29**(6):555–565.
20. Tai KX, Carver RH, Obaid EE. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering* 1991; **17**(1):45–63.
21. Vetter JS, Supinski BR. Dynamic software testing of MPI applications with Umpire. *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE Press (electronic publication): New York, 2000.
22. Frankl FG, Weyuker EJ. Data flow testing in the presence of unexecutable paths. *Workshop on Software Testing*, Banff, Canada, July 1986; 4–13.
23. Krawczyk H, Wiszniewski B. Classification of software defects in parallel programs. *Technical Report 2*, Faculty of Electronics, Technical University of Gdansk, Poland, 1994.
24. Howden WE. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering* 1976; **2**:208–215.
25. Tanenbaum AS. *Modern Operating Systems* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 2001.
26. Souza SRS, Vergilio SR, Souza PSL, Simão AS, Bliscosque TG, Lima AM, Hausen AC. Valipar: A testing tool for message-passing parallel programs. *International Conference on Software Knowledge and Software Engineering* (*SEKE05*), Taipei, Taiwan, 2005; 386–391.
27. Simão AS, Vincenzi AMR, Maldonado JC, Santana ACL. A language for the description of program instrumentation and the automatic generation of instrumenters. *CLEI Electronic Journal* 2003; **6**(1).
28. Chusho T. Test data selection and quality estimation based on concept of essential branches for path testing. *IEEE Transactions on Software Engineering* 1987; **13**(5):509–517.
29. Chaim MJ. Poketool—uma ferramenta para suporte ao teste estrutural de programas baseados em anlise de fluxo de dados. *Master's Thesis*, DCA/FEE/UNICAMP, Campinas, SP, 1991.
30. Vergilio SR, Maldonado JC, Jino M. Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction. *Journal of the Brazilian Computer Society* 2006; **12**(1).
31. Wong WE, Lei Y, Ma X. Effective generation of test sequences for structural testing of concurrent programs. *Tenth IEEE International Conference on Engineering of Complex Systems* (*ICECCS'05*), 2005; 539–548.
32. Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 1986; **8**(2):244–263.
33. Siegel SF, Mironova A, Avrunin GS, Clarke LA. Using model checking with symbolic execution to verify parallel numerical program. *International Symposium on Software Testing and Analysis*, 2006; 157–167.
34. Siegel SF. Model checking nonblocking MPI programs. *Verification*, *Model Checking and Abstract Interpretation* (*Lecture Notes in Computer Science*, vol. 4349). Springer: Berlin, 2007; 44–58.
35. Matlin OS, Lusk E, McCune W. Spinning parallel systems software. *SPIN* (*Lecture Notes in Computer Science*, vol. 2318). Springer: Berlin, 2002; 213–220.
36. Pervez S, Gopalakrisman G, Kirby RM, Thakur R, Gropp W. Formal verification of programs that use mpi one-sided communication. *PVM/MPI* (*Lecture Notes in Computer Science*, vol. 4192). Springer: Berlin, 2006; 30–39.
37. The LAM/MPI Team. *XMPI*. Open Systems Laboratory, Indiana University, Bloomington, IN.
38. Krawczyk H, Kuzora P, Neyman M, Proficz J, Wiszniewski B. STEPS—A tool for testing PVM programs. *Third SEIHPC Workshop*, January 1998. Available at: http://citeseer.ist.psu.edu/357124.html.

Apêndice

D

(Brito et al. 2013)Brito, M. A. S.; Souza, S. R. S. An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs. In: ICCS - International Conference on Computational Science, Barcelona, Espanha, p. 250-259,2013.

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 250 – 259

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

An Empirical Evaluation of the Cost and Effectiveness of Structural Testing Criteria for Concurrent Programs

Maria A. S. Brito[a], Simone R. S. Souza[a], Paulo S. L. Souza[a,*]

[a]*Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC/USP, P.O. 668, São Carlos (SP), Brazil, 13560-970*

Abstract

Concurrent program testing is not a trivial task. Features like nondeterminism, communication and synchronization impose new challenges that must be considered during the testing activity. Some initiatives have proposed testing approaches for concurrent programs, in which different paradigms and programming languages are considered. However, in general, these contributions do not present a well-formed experimental study to validate their ideas. The problem is that the data used and generated during the validation is not always available, hampering the replication of studies in the context of other testing approaches. This paper presents an experimental study, taking into account the concepts of the Experimental Software Engineering to evaluate the cost, effectiveness and strength of the structural testing criteria for message-passing programs. The evaluation was conducted considering a benchmark composed of eight MPI programs. A set of eight structural testing criteria defined for message-passing programs was evaluated with the ValiMPI testing tool, which provides the support required to apply the investigated testing criteria. The results indicate the complementary aspect of the criteria and the information about cost and effectiveness has contributed to the establishment of an incremental testing strategy to apply the criteria. All material generated during the experimental study is available for further comparisons.

Keywords: Software testing; Concurrent programs; MPI programs; Experimental study

1. Introduction

The demand for distributed and parallel applications has been growing due to the advanced hardware technology, which provides more efficient machines and allows to process large volumes of data. However, these applications are inevitably more complex than sequential ones. Every concurrent software contains features, such as nondeterminism, synchronization and inter-process communication, which significantly hamper their validation and their testing. Approaches to test concurrent programs efficiently and effectively have been proposed and are an important factor for the success and quality of the programs built in this domain.

Several studies have been conducted to define approaches to test concurrent programs [1, 2, 3, 4, 5]. In general, these studies present some evaluation to demonstrate the applicability of their contribution. The problem is that the data used is not always available, hampering the replication of the studies and the fair comparison among different testing techniques for concurrent programs. In the context of sequential program testing, Weyuker [6] pointed out the importance of comparative studies to evaluate different testing techniques, allowing the replication

*Corresponding author. Tel.: +55-16-3373-9700 ; fax: +55-16-3373-9700
E-mail address: masbrit@icmc.usp.br; srocio@icmc.usp.br; pssouza@icmc.usp.br.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.188

of the experimental studies. Thus, demonstrating the applicability and effectiveness of testing techniques is as important as proposing testing techniques for new application domain.

The empirical evaluation of techniques, criteria and testing tools has been intensified in recent years, mainly in the context of traditional programs. These empirical evaluations, in general, consider three basic factors for a comparison: cost, effectiveness and strength [7]. Cost refers to the effort required to satisfy a testing criterion and can be measured by the number of test cases necessary to cover it. Effectiveness refers to the ability of a test set to reveal defects. Strength refers to the probability to satisfy a testing criterion using a test set adequate to another testing criterion. These comparison factors are important for the proposition of an efficient testing strategy taking into account the benefits of each testing criterion.

This paper has contributed in this direction, presenting an experimental study that evaluates structural testing criteria defined for message-passing programs, analyzing their cost, effectiveness and strength. Experimental study takes into account the process defined by Wholin [8], which includes activities for the definition, planning, conduction, analysis and packing of experimental studies. A benchmark composed of eight MPI (Message Passing Interface) programs is defined and used in our study. MPI is a message-passing library interface specification for the development of portable message-passing concurrent programs using sequential languages, such as C and Fortran [9]. Our work considers MPI programs written in C language.

The analyzed testing criteria were proposed by Souza et al. [10] and include structural criteria to test concurrent and sequential aspects of message-passing programs. Information about control, data and communication flows is extracted from a program under test and used to guide the generation of a test case set. Eight different testing criteria were analyzed using the ValiMPI support tool [11]. This tool provides the required resources to apply test cases and evaluate their coverage in programs considering the structural testing criteria for MPI programs.

The material generated during the experimental study, including programs, results of testing activity, ValiMPI tool and results of experimental study has been organized and is available for public access, providing relevant information to further studies and comparisons. According to our knowledge, it is the first study which uses concepts of the Experimental Software Engineering for the definition, conduction and analysis of the testing criteria in the context of concurrent programs.

The remaining of the paper is organized as follows: Section 2 presents the test model and the structural testing criteria for message-passing programs, as well as the ValiMPI testing tool. Section 3 describes the experimental study, including the definition, planning, results and analysis. Section 4 reports the related works and Section 5 presents the final considerations and future research directions.

2. Structural Testing Criteria for Message-Passing Programs

In message-passing programs, communication is made by *send* and *receive* basic primitives, in which a process can send a message to another process or to a group of processes. The first one is called point-to-point communication and the second is called collective communication. In both cases, the test activity must establish an association between the variables sent and the location where these variables are used in the receiver process(es). Besides, it is important to define a strategy to derive all possible synchronizations among the processes of the concurrent software processes. Executing these distinct synchronizations allows the verification of different pairs of definition and use of variables in different processes. Associated to these synchronizations there are important questions that must be considered, such as non-determinism, controlled execution and race conditions.

Souza et al. [10] defined a test model and a set of structural testing criteria that represent the main features of message-passing programs, including control, data and communication aspects. The test model considers that a concurrent program is a set $Prog = p^0, p^1, ..., p^{n-1}$ composed of its n parallel processes. A Control Flow Graph (CFG) of each process p is generated using the same concepts of sequential programs. Each CFG^p represents the control flow of a process p. A Parallel Control Flow Graph (PCFG) generated for $Prog$ is composed of CFG^p (to $p = 0 \ldots n-1$) and edges of communication among processes. N represents the set of nodes and E represents the set of edges in PCFG. E has two subsets: E_p, which are edges of a same process and E_s, composed of edges that represent the communication between processes, called interprocess edges. A node i in a process p is represented by notation n_i^p. Two subsets of N are defined: N_s, which are the nodes composed of primitives *send* and N_r, which are nodes composed of primitives *receive*. A set R_i^p is associated with each $n_i^p \in N_s$ containing possible nodes that can receive the message sent by node n_i^p. This set is important to establish all possible communication pairs.

A path π in CFG^p is called intraprocess if it contains no interprocess edges and is composed of a sequence of nodes $\pi = (n_1, n_2, ..., n_m)$ in which $(n_i, n_{i+1} \in E_i^p)$. A path π that contains at least one interprocess edge is called an interprocess path.

In relation to data flow information, the concepts employed in sequential programs are considered and extended to include the communication between processes. Thus, in addition to the computational and predicative use of variables (c-use and p-use), the communication use (s-use) is defined and associated with the variables in a sent message. Considering these concepts, the following associations between definition and use of variables are defined:

a) *s-use association*: defined by a triple $(n^{p1}, (m^{p1}, k^{p2}), x)$, such that $x \in def(n^{p1})$ and (m^{p1}, k^{p2}) has an s-use of x and there is a definition-clear path with respect to x from n^{p1} to (m^{p1}, k^{p2}).

b) *s-c-use association*: given by $(n^{p1}, (m^{p1}, k^{p2}), l^{p2}, x^{p1}, x^{p2})$, where there is an s-use association $(n^{p1}, (m^{p1}, k^{p2}), x^{p1})$ and a c-use association (k^{p2}, l^{p2}, x^{p2}).

c) *s-p-use association*: given by $(n^{p1}, (m^{p1}, k^{p2}), (n^{p2}, m^{p2}), x^{p1}, x^{p2})$, in which there is an s-use association $(n^{p1}, (m^{p1}, k^{p2}), x^{p1})$ and a p-use association $(k^{p2}, (n^{p2}, m^{p2}), x^{p2})$.

Based on the information derived from the test model, a family of structural testing criteria for message-passing programs was defined in details in Souza et al. [10]. The following testing criteria are considered in our experimental study:

1. **all-nodes-s criterion (ans)**: the test sets must execute paths that cover all nodes $n_i^p \in N_s$.
2. **all-nodes-r criterion (anr)**: the test sets must execute paths that cover all nodes $n_i^p \in N_r$.
3. **all-nodes criterion (an)**: the test sets must execute paths that cover all nodes $n_i^p \in N$.
4. **all-edges-s criterion (aes)**: the test sets must execute paths that cover all edges $(n_j^{p1}, n_k^{p2}) \in E_s$.
5. **all-edges criterion (ae)**: the test sets must execute paths that cover all edges $(n_j, n_k) \in E$.
6. **all-c-uses criterion (acu)**: the test set must execute paths that cover all c-use associations.
7. **all-p-uses criterion (apu)**: the test set must execute paths that cover all p-use associations.
8. **all-s-uses criterion (asu)**: the test set must execute paths that cover all s-use associations.

The ValiMPI tool supports the application of these testing criteria, providing functionalities to generate a test session, save and run test data and evaluate the testing coverage for a given criterion [11]. The ValiMPI extracts control, data and synchronization information from a source code of the program to generate the required elements of each testing criterion. The program is instrumented by inserting *check-points* that allow to generate an execution trace, in which it is possible to evaluate the coverage obtained by test cases in relation to testing criteria. Functionalities are available to execute a test case with all the possible synchronization sequences, assuring their coverage during the test activity.

3. Experimental Study

Considering the objective of this experimental study, the following goals were defined based on guidelines for Experimental Software Engineering proposed by Wohlin [8]:

- **Object of study**: the testing criteria for message-passing programs all-nodes, all-nodes-r, all-nodes-s, all-edges, all-edges-s, all-c-uses, all-p-uses and all-s-uses;
- **Purpose**: evaluation of the application cost, effectiveness and strength of each testing criterion;
- **Quality focus**: cost, effectiveness and strength of each testing criterion;
- **Perspective**: viewpoint of the researcher;
- **Context**: this experimental study was carried out by us considering a set of programs of calculus, physics and bioinformatics, using ValiMPI testing tool to support the application of the analyzed testing criteria.

3.1. Planning

We selected a benchmark composed of eight representative programs, seven of them are classical problems in concurrent programming and one of them is a real concurrent program from bioinformatics domain, proposed by Bonetti et al. in [12]. The eight programs are:

1. **GCD**: this program calculates the greatest common divisor considering three numbers, according to algorithm presented in [13].
2. **Jacobi**: this program implements the iterative method of Gauss–Jacobi for solving a linear system of equations.
3. **Mmult**: this program implements the multiplication of a matrix using domain decomposition, according to algorithm described in [14].
4. **Qsort**: this program implements the recursive quicksort method, according to algorithm presented in [15].
5. **Reduction**: this program implements the reduction operation of distributed data.
6. **Sieve**: this program implements the sieve of eratosthenes, according to algorithm described in [14].
7. **Trap**: this program calculates an approximation of the integral of a function x via trapezoidal method.
8. **Van der Waals**: real application developed by Bonetti et al. in [12], which calculates the Van der Waals energy of a protein using concepts of genetic algorithms.

These programs were implemented in *C/MPI* and, in order to standardize the size of these programs, each one is composed of four parallel processes. Information about the complexity of each program is presented in Table 1:

Table 1. Characteristics of the case studies.

Program	LOC	Send Primitives	Receive Primitives
GCD	112	5	5
Jacobi	691	11	19
Mmult	192	9	15
Qsort	480	7	13
Reduction	132	1	1
Sieve	113	3	7
Trap	77	1	1
Waals	540	4	4

3.1.1. Hypotheses

Considering the objectives of this study, we defined the research question and a set of hypotheses, which were used to analyze the results.

Research question*: what are the effectiveness, application cost and strength of the testing criteria for message-passing programs?*

Null Hypothesis 1 (NH1): the application cost is the same for all structural testing criteria analyzed.

Alternative Hypothesis 1 (AH1): the application cost is different for at least one structural testing criterion for message-passing programs.

Null Hypothesis 2 (NH2): the effectiveness is the same for all structural testing criteria analyzed.

Alternative Hypothesis 2 (AH2): the effectiveness is different for at least one structural testing criterion for message-passing programs.

Null Hypothesis 3 (NH3): in relation to strength, no testing criterion subsumes another.

Alternative Hypothesis 3 (AH3): there is at least one testing criterion that subsumes another.

3.1.2. Variables Selection

In this section we present the dependent and independent variables that are used to represent the treatments of the experiment. The independent variables are the input of the experiment and are manipulated and controlled. The dependent variables are the response of the independent variables and represent the effect of the changes in the independent variables.

In this study, the independent variables of interest are: a) structural testing criteria for concurrent programs, b) programs used; and c) faulty programs.

In relation to dependent variables, the following variables are of interest: a) number of required elements of each testing criterion; b) size of the test set adequate to each testing criterion; c) number of infeasible elements of each testing criterion; d) number of defects revealed by each testing criterion; and e) percentage of coverage obtained to satisfy each testing criterion.

3.2. Preparation of the Experiment

Some tasks were carried out before the execution of the experiment, including: preparation of the programs, definition of set of defects to be considered, definition of how the defects would be inserted in the programs and installation of the ValiMPI testing tool.

The environment for the development of the experimental study has the following features: GNU/Linux operational system using the distribution Ubuntu 10.04 with the kernel 2.6.32, compiler *gcc* 4.1.2, Open MPI 1.4 and ValiMPI testing tool.

3.3. Execution of the Experiment

Initially, adequate test sets were manually generated for the programs, where each test set traverses all feasible required elements of a particular testing criterion. The programs were executed with the test sets using the ValiMPI tool and observing the coverage obtained. New test cases were added until all feasible required elements had been executed. In this phase, the infeasible elements were manually identified. The cost was evaluated considering the size of the adequate test sets. Also, information about the quantity of infeasible required elements were used to calculate the cost of each testing criterion.

To evaluate the effectiveness, defects were manually inserted into the programs, based on classifications of defects proposed by DeSouza et al. [16] and Agrawal et al. [17]. These classifications are based on errors made by developers, being relevant for our experimental study. Following a strategy similar to mutation testing, the defects were systematically inserted in each program, generating one program version for each inserted defect, totaling 334 faulty programs. Table 2 shows the number of defects inserted in each program, according to the type of defect considered. Some kind of defects did not generate faulty versions for some programs. Due to the features of each program, some defects did not make sense to be applied or the defects have generated an erroneous version, for instance, a deadlock in the program. In this case, the faulty versions were not considered.

These faulty programs were executed with the adequate test set and the ability to reveal the defects was registered.

The adequate test sets were also used to evaluate the strength of the criteria, in which a test set T_{c1}, adequate to criterion $C1$, is evaluated in relation to criterion $C2$; if T_{c1} is adequate to $C2$, then $C1$ may include $C2$. Thus, if the coverage of T_{c1} to $C2$ is low, this means that the criterion $C1$ has higher strength otherwise, $C1$ has lower strength in relation to $C2$. In this phase only the original (correct) programs were considered.

Table 2. Quantity of defects inserted in each program.

Type of defect	GCD	Jacobi	Mmult	Qsort	Reduction	Sieve	Trap	Waals
incorrect loop or selection structure	2	8	0	2	3	1	2	1
incorrect process in messages	2	4	1	0	0	2	1	0
source process changed by ”any” process in messages	1	5	0	2	0	1	0	0
incorrect size of array	2	4	3	1	0	4	0	0
non initialized variable	0	5	4	0	3	1	4	2
incorrect data types	0	0	0	0	0	0	0	3
incorrect size of message	0	2	4	1	0	2	4	2
incorrect message address	2	4	2	4	4	6	3	4
incorrect type of parameter	1	3	3	1	1	1	1	0
incorrect message data type	1	1	1	1	0	1	1	2
replacement blocking by non-blocking message	5	1	1	2	2	4	1	2
change of operator in the variable definition	2	8	5	3	2	8	7	5
incorrect data sent or received	1	5	5	3	2	5	3	4
change of the logical operator in predicative statements	2	5	3	5	1	7	2	4
missing statements	5	8	6	4	2	7	5	5
incorrect variable definition	4	4	4	5	1	7	0	5
increment/decrement of variables in messages	0	4	2	1	0	3	0	0
Total	**30**	**71**	**44**	**35**	**22**	**60**	**33**	**39**

3.4. Analysis of Results

This section presents some analysis related to the hypotheses formulated, where the results of statistical tests admitted a 95% significance level. For space reasons, some results are not presented here; more details and all the material used in the experiment can be obtained in: www.labes.icmc.usp.br/~experiments/MPIConcurrentCriteria.rar.

3.4.1. Data Analysis - Cost

Hypotheses $NH1$ and $AH1$, related to the application cost, were evaluated considering number of required elements, number of infeasible elements and size of the adequate test set.

Figure 1 shows the boxplot of the cost, considering the size of adequate test sets. A boxplot graphically represents five aspects of a data distribution (from the bottom to the top): the smallest observation, lower quartile, median, upper quartile, and largest observation. It can also indicate outliers for some data of the distribution (see points in boxplot in Figure 2). We can observe that the medians are higher for the testing criteria *all-s-uses (asu)*, *all-p-uses (apu)* and *all-edges (ae)*, indicating that the number of test cases necessary to satisfy these criteria is larger than for the other criteria.

The ShapiroWilk indicated that the population is distributed normally, thus the ANOVA methods was employed to this statistical analysis. We have obtained a *p-value* < 0.05 and the hypothesis $AH1$ is accepted indicating that at least one testing criterion presents a different value to the cost against the other criteria. Thus, we have applied the Tukey test for multiple comparisons to verify which criteria have significant difference in cost. The results indicate a significant difference between the following pairs of criteria: 1) *all-s-uses* and *all-nodes-r* and 2) *all-s-uses* and *all-nodes-s*. This result is according to the boxplot in Figure 1 demonstrating that the *all-s-uses* is the costlier criterion and the *all-nodes-r* and *all-nodes-s* are the lower cost criteria.

Considering required elements and infeasible elements, the results of the cost evaluation are similar and, therefore hypothesis $AH1$ can be accepted.

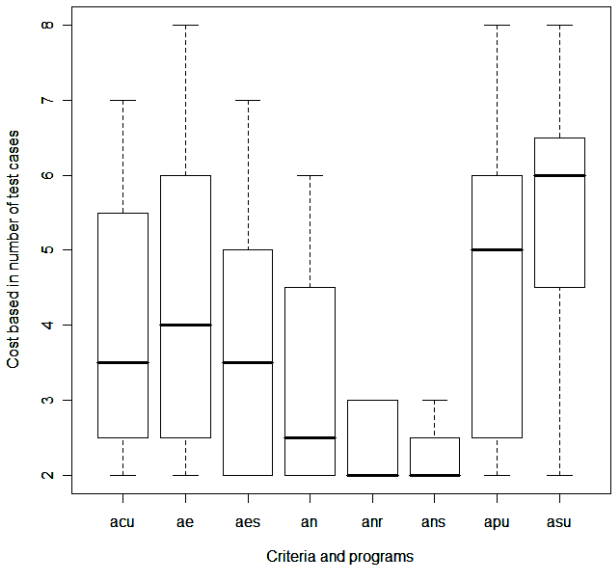


Fig. 1. Boxplot of the cost based on the size of adequate test. sets

3.4.2. Data Analysis - Effectiveness

In relation to effectiveness, the hypotheses $NH2$ and $AH2$ were evaluated (Figure 2). The boxplot in Figure 2 shows evidences that some criteria are more effective than others. For instance, the *all-s-uses (asu)* criterion is able to reveal 100% of defects injected in the programs, except for one program, whose effectiveness was 95.8% (outlier). They also show that the medians are higher for the testing criteria *all-p-uses (apu)*, *all-c-uses (acu)* and *all-edges (ae)*.

An important result is that the *all-edges* is an effective criterion to reveal faults in context of concurrent programs, contradicting the results obtained when this criterion is considered for sequential programs. This result became interesting because the cost of this criterion is lower, motivating further investigation of this testing criterion.

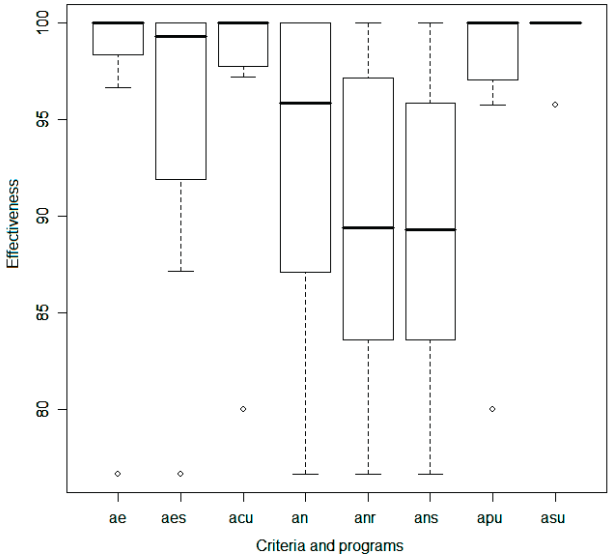


Fig. 2. Boxplot for the effectiveness of the testing criteria.

3.4.3. Data Analysis - Strength

In relation to strength, the hypotheses $NH3$ and $AH3$ were evaluated using the statistical method of cluster analysis. The hierarchical cluster analysis splits a sample in groups based on similarities between them. A dendrogram is a technique of the cluster analysis and it groups the data using hierarchical trees and measurements as the average, for example. In this sense, we have applied this analysis for each testing criterion and we have generated a dendrogram for each testing criterion studied.

Figure 3 presents the dendrogram chart for *all-s-uses (asu)* criterion, in which the adequate test set to this criterion was applied to all the other criteria. As result, all testing criteria in the same level of *all-s-uses* criterion obtained 100% coverage by the adequate test set to this criterion. This happens for the criteria *all-nodes (an)*, *all-nodes-s (ans)* and *all-nodes-r (anr)*. Thus, in this experimental study the *all-s-uses (asu)* criterion includes the criteria *all-nodes-s (ans)*, *all-nodes (an)* and *all-nodes-r (anr)*.

One dendrogram chart as been made for each testing criteria and the following results regarding strength were observed:

- *all-nodes* includes *all-nodes-r* and *all-nodes-s*;
- *all-nodes-s* includes *all-nodes-r* (and vice-versa);
- *all-edges* includes *all-nodes*, *all-nodes-r*, *all-nodes-s* and *all-edges-s*;
- *all-edges-s* includes *all-nodes-r* and *all-nodes-s*;
- *all-p-uses* includes *all-nodes*, *all-nodes-r* and *all-nodes-s*.

Based on these results, the null hypothesis $NH3$ can be rejected and the alternative hypothesis $AH3$ is accepted, indicating that the criteria can be complementary.

3.5. Discussion of Results

It is desirable that a test strategy presents a high effectiveness with a low cost and, therefore, these measures are strongly related. Thus, a testing criterion with high effectiveness can be prohibitive to apply if the cost is expensive. Leading in consideration the results obtained, we have proposed a test strategy to apply these testing criteria.

Figure 4 illustrates a partial order to apply the criteria. We have suggested that the tester starts the tests with the criteria *all-s-uses (a-s-u)*, *all-p-uses (a-p-u)* or *all-c-uses (a-c-u)*. From one of these criteria, the others can be

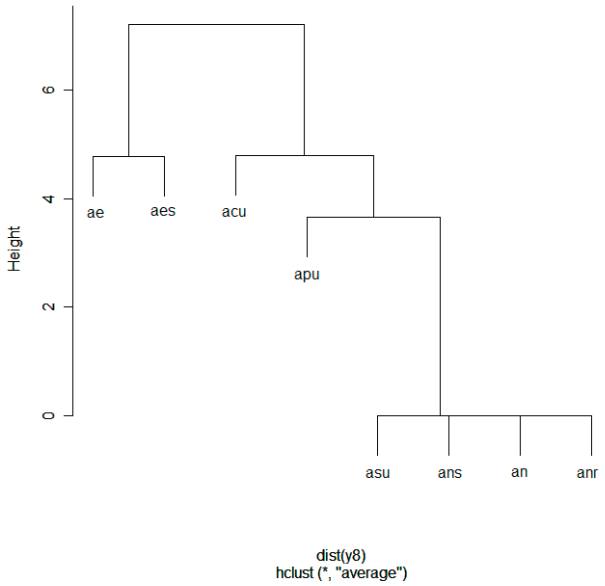


Fig. 3. Strength of the all-s-uses criterion.

added according to the flow suggested in the Figure 4. An important aspect is that each testing criterion considers different aspects for the tests, such as use of variables (*all-p-uses, all-c-uses*), communication between processes (*all-s-uses, all-edges-s, all-nodes-r, all-nodes-s*) or sequential control flow (*all-edges, all-nodes*). Using the order presented in Figure 4, the tester can choose the testing criteria according to which information he/she desires to cover in the concurrent program, improving the quality of the test activity.

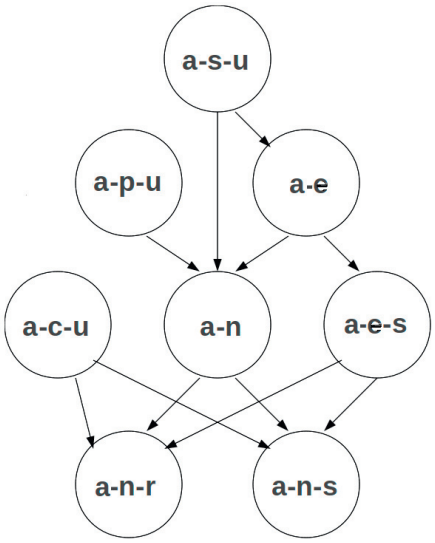


Fig. 4. A possible order to apply the testing criteria.

3.6. Threats to Validity

As for every empirical study, it is important to identify the threats to the validity of the reported results. Based on the principles presented by Wohlin [8] we have identified the following threats for our study:

Construction Validity: the ValiMPI testing tool automatically generates measurement about the testing criteria, avoiding the threat of satisfaction of some hypothesis or a particular criterion favored by the researcher.

Internal Validity: a factor that can compromise this validity is the bias that may occur during the insertion of defects in the programs. As the researcher knows the testing criteria, he could insert defects that would be easily revealed by the criteria. This threat was mitigated by the application of classifications of defects presented in the literature, in which the defects were inserted following a strategy similar to mutation testing strategy. Besides, the programs used in the study were not developed by the researcher involved.

Conclusion Validity: in this study the researcher might influence the results by trying to obtain the specific result. This threat was mitigated by the application of statistical methods to verify the normality of the data set, applying the methods according to the data set distribution.

4. Related Work

Recently, there has been significant progress in the understanding of the strength and limitations of concurrent programs testing. However, there have been few contributions concerned with experimental studies to evaluate testing criteria or compare testing tools in this context.

In [18, 19, 20] the authors evaluate the reachability testing and VeriSoft and RichTest tools and in [21] a technique for systematic exploration of thread interleavings is presented. The authors describe experiments to evaluate the technique.

In [22] a new technique for bugs finding in concurrent programs, called race-directed random testing (or RaceFuzzer) is proposed. The experimental results demonstrate that RaceFuzzer is able to create real situations of race conditions with very high probability to find them.

An experimental study to evaluate an abstraction-guided symbolic execution technique that detects errors in concurrent programs, is described in [23]. Five multi-threaded Java programs were used in the study. The results show that the technique generates feasible execution paths and finds concurrency errors quickly when compared to exhaustive symbolic execution testing.

In a similar study, we have investigated the same factors for other testing criteria, proposed for shared-memory programs [24]. Considering similar hypotheses, we have analysed the cost, effectiveness and strength of the testing criteria for multithreaded programs, implemented in (POSIX Threads). Differently, in this study we have compared sequential testing criteria and concurrent testing criteria in order to observe which group presents best effectiveness and minor cost. The results indicate that the concurrent testing criteria present a minor cost and are able to reveal some kind of defects, which are not revealed for sequential testing criteria.

In the context of MPI programs we have not identified studies that propose other structural testing criteria. In [25] the authors used formal methods that are very different from the experimental study presented here.

5. Conclusions and Future Works

This paper presented an experimental study to evaluate a family of structural testing criteria for message-passing programs. The objective was finding evidences about the cost, effectiveness and strength of those testing criteria, in order to define a testing strategy to apply them. The study was conducted considering the Experimental Software Engineering Process defined by Wholin [8].

The results show that the cost of the testing criteria based on control and communication flows is in general lower than the cost of testing criteria based on data flow and message-passing. These results are similar to those for sequential programs. In relation to effectiveness to reveal faults, the data analysis indicates that the criteria *all-s-uses* and *all-edges* are effective to reveal faults. Also, we have observed that some faults are only revealed for some criteria, indicating that criteria with minor effectiveness are relevant to reveal specific defects.

In relation to strength analysis, the *all-s-uses* is the strongest criterion, may include other criteria. The results of the strength indicate the complementary relationship among the testing criteria, therefore we have proposed an application test strategy.

As part of future work, we intend to compare our testing criteria with other testing approaches, for instance, model-check or mutation testing. Considering the recent work about mutation testing for MPI programs, presented

in [26], we intend to use the mutation testing to generate the faulty program versions in order to evaluate the effectiveness of our testing criteria.

According to Wohlin et al. [8], as an experiment will never provide a final answer to a question, it is important to facilitate its replication. In this sense, one of the contributions of this study is the packing of the material generated in the experiment. This lab package is available for public access, allowing a further evaluation, considering other testing mechanisms for message-passing programs.

Acknowledgements

The authors would like to acknowledge the Brazilian funding agency FAPESP for the financial support, under process 2009/04517-2.

References

[1] R. Taylor, D. Levine, C. Kelly, Structural testing of concurrent programs, IEEE Trans. on Software Engineering 18 (3) (1992) 206–215. doi:10.1109/32.126769.
[2] R.-D. Yang, C.-G. Chung, Path analysis testing of concurrent programs, Inf. Softw. Technol. 34 (1) (1992) 43–56.
[3] W. Wong, Y. Lei, Reachability graph-based test sequence generation for concurrent programs, International Journal of Software Engineering and Knowledge Engineering 18 (6) (2008) 803–822.
[4] R. Carver, Y. Lei, A general model for reachability testing of concurrent programs, International Conference on Formal Engineering Methods 3308 (2004) 76–98.
[5] Y. Lei, R. H. Carver, R. Kacker, D. Kung, A combinatorial testing strategy for concurrent programs, Software Testing Verification and Reliability 17 (4) (2007) 207–225.
[6] E. J. Weyuker, The cost of data flow testing: An empirical study, IEEE Trans. Softw. Eng. 16 (2) (1990) 121–128.
[7] W. E. Wong, A. P. Mathur, J. C. Maldonado, Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness, in: Software Quality and Productivity: Theory, practice and training, Chapman & Hall, Ltd., London, UK, UK, 1995, pp. 258–265.
[8] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering: an introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.
[9] M. P. I. Forum, Mpi: A message-passing interface standard - version 2.2, Tech. rep., University of Tennessee, Knoxville, Tennessee (2009).
[10] S. R. S. Souza, R. Vergilio, P. S. L. Souza, S. Simão, A. C. Hausen, Structural testing criteria for message-passing parallel programs, Concurrency Computation Practice and Experience. 20 (16) (2008) 1893–1916.
[11] A. C. Hausen, S. R. Vergilio, S. R. S. Souza, P. S. L. Souza, A. S. Simão, A tool for structural testing of MPI programs, in: 8th IEEE Latin-American Test Workshop, Cuzco, Lima, 2007.
[12] D. R. F. Bonetti, A. C. B. Delbem, G. Travieso, P. S. L. Souza, Optimizing van der waals calculi using cell-lists and MPI, in: IEEE Congress on Evolutionary Computation, IEEE, 2010, pp. 1–7.
[13] H. Krawczyk, B. Wiszniewski, Classification of software defects in parallel programs, Tech. rep., Faculty of Eletronics, Technical University of Gdansk, Poland (1994).
[14] M. J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Education Group, 2003.
[15] A. Grama, G. Karpys, V. Kumar, A. Gupta, Introduction to parallel computing, 2nd Edition, Addison Wesley, 2003.
[16] J. DeSouza, B. Kuhn, B. R. Supinski, V. Samofalov, S. Zheltov, S. Bratanov, Automated, scalable debugging of MPI programs with intel message checker, in: International Workshop on Software Engineering for High Performance Computing System Applications, New York, NY, USA, 2005, pp. 78–82.
[17] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, E. H. Spafford, Design of mutant operators for the C programming language, Tech. rep., Software Engineering Research Center/Purdue University, sERC-TR-41-P (1989).
[18] C. R. Lei, Y., A new algorithm for reachability testing of concurrent programs, in: 16th IEEE International Symposium on Software Reliability Engineering, 2005, pp. 10–355.
[19] C. R. Lei, J., A stateful approach to testing monitors in multithreaded programs, in: 12th IEEE International Symposium on Higth-Assurance Systems Engineering, 2010, pp. 54–63.
[20] R. H. Carver, Y. Lei, Distributed reachability testing of concurrent programs, Concurr. Comput. : Pract. Exper. 22 (18) (2010) 2445–2466.
[21] P. Fonseca, C. Li, R. Rodrigues, Finding complex concurrency bugs in large multi-threaded applications, in: Conference on Computer Systems, EuroSys '11, New York, NY, USA, 2011, pp. 215–228.
[22] K. Sen, Race directed random testing of concurrent programs, SIGPLAN Not. 43 (6) (2008) 11–21.
[23] N. Rungta, E. G. Mercer, W. Visser, Efficient testing of concurrent programs with abstraction-guided symbolic execution, in: International SPIN Workshop on Model Checking Software, Berlin, Heidelberg, 2009, pp. 174–191.
[24] S. M. Melo, S. R. S. Souza, P. S. L. Souza, Structural testing for multithreaded programs: An experimental evaluation of the cost, strength and effectiveness, in: International Conference on Software Engineering and Knowledge Engineering, San Francisco, USA, 2012, pp. 476–479.
[25] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, R. M. Kirby, ISP: a tool for model checking MPI programs, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, New York, NY, USA, 2008, pp. 285–286.
[26] R. A. Silva, S. R. S. Souza, P. S. L. Souza, Mutation testing for concurrent programs in MPI, in: 13th Latin American Test Workshop, Quito, Ecuador, 2012, pp. 69–74.

Apêndice

E

(Souza et al. 2011d) Souza, S. R. S.; Souza, P. S. L.; Machado, M. C. C.; Camillo, M. S.; Simao, A. S.; Zaluska, E. Using coverage and reachability testing to improve concurrent program testing quality. In: 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE2011), Miami Beach, USA, 2011d, p. 207-212.

PROCEEDINGS

SEKE 2011

The 23rd International Conference on Software Engineering & Knowledge Engineering

Sponsored by

Knowledge Systems Institute Graduate School, USA

Technical Program

July 7-9, 2011

Eden Roc Renaissance Miami Beach, Florida, USA

Organized by

Knowledge Systems Institute Graduate School

Semantic Web Technologies

Software Testing and Debugging

Using Coverage and Reachability Testing to Improve Concurrent Program Testing Quality

Simone R. S. Souza[1], Paulo S. L. Souza[1], Mario C. C. Machado[1],
Mário S. Camillo[1], Adenilso Simão[1] and Ed Zaluska[2]
[1] Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo
P.O. 668 – São Carlos – Brasil – 13560-970
{srocio, pssouza,mmachado, adenilso}@icmc.usp.br, mariocamillo@gmail.com
[2] Eletronics and Computer Science – University of Southampton
ejz@ecs.soton.ac.uk

Abstract

The testing of concurrent software is a challenging task. A number of different research approaches have investigated adaptation of the techniques and the criteria defined for sequential programs. A major problem with the testing of concurrent software that persists is the high application cost due to the large number of the synchronizations that are required and that must be executed during testing. In this paper we propose a complementary approach, using reachability testing, to guide the selection of the tests of all synchronization events according to a specific coverage criterion. The key concept is to take advantage of both coverage criteria, which are used to select test cases and also to guide the execution of new synchronizations, and reachability testing, which is used to select suitable synchronization events to be executed. An experimental study has been conducted and the results indicate that it is always advantageous to use this combined approach for the testing of concurrent software.

1. Introduction

Concurrent applications are inevitably more complex than sequential ones and, in addition, all concurrent software contains features such as nondeterminism, synchronization and inter-process communication which significantly increase the difficulty of validation and testing.

For sequential programs, many testing problems were simplified with the introduction of testing criteria and the implementation of supporting tools. A testing criterion is a predicate to be satisfied by a set of test cases which can be used as a template for the generation of test data [10].

Extending previous work on sequential program structural testing criteria, we have proposed structural testing criteria for the validation of concurrent programs, applicable to both message-passing software [14] and shared-memory software [12]. These testing criteria are designed to exploit information about the control, data and communication flows of concurrent programs, considering both sequential and parallel aspects.

The use of these criteria significantly improves the quality of the test cases, providing a coverage measure that can be used in two important testing procedures. In the first one, the criteria can be used to guide the generation of test cases, where the criteria are used as guideline for test data selection. The second testing procedure is related to the evaluation of a test set; in this case, the criteria can be used to determine when the testing activity can be terminated based on sufficient coverage of the required elements. The main contribution of the proposed testing criteria is to provide an efficient coverage measure for evaluating the progress of the testing activity and the quality of test cases.

This approach uses static analysis of the program under test to extract relevant information for testing, which is straightforward to apply and generate relevant information for coverage testing. The problem is the large number of infeasible elements generated that must be analyzed. An element is infeasible if there is no set of values for the parameters (the input and global variables) that cover that element. Complete determination of infeasible elements is an extremely difficult problem and it is not possible to determine them automatically.

Lei and Carver [7] proposed a method (based on reachability testing) to obtain all of the executable synchronizations of a concurrent program (from a given execution of the program) in a way that reduces the number of redundant synchronizations. As this method uses dynamic information, only feasible synchronizations are generated, which

is a considerable advantage. However, a difficulty with this method is the high number of possible combinations of synchronization that are generated. For complex programs, this number is very high, which limits the practical application of this approach. In [1], Carver and Lei proposed a distributed reachability testing algorithm, allowing different test sequences be executed concurrently. This algorithm reduces the time to execute the synchronizations, but the authors do not comment about the effort necessary to analyze the results from these executions.

Lei and Carver's [7] method is essentially complementary to our approach. They do not address how to select the test case which will be used for the initial run, while we use the static analysis of the program to select the optimum test cases in advance.

In this paper we propose a complementary approach, using reachability testing to target coverage testing for synchronization events. The idea is to take appropriate advantage of both approaches: information about synchronizations provided by the coverage criterion are used to decide which race variants will be executed, selecting only synchronizations that have not already been covered by existing test cases. It is therefore possible to execute each synchronization at least once and to use reachability testing to select only those synchronizations that are feasible.

This paper is structured as follows. In Section 2 we describe related work on the testing of concurrent software, presenting more details on the coverage testing and reachability testing approaches. In Section 3 we present the test strategy proposed in this paper. In Section 4, an experimental study to evaluate our test strategy is presented and the results obtained are discussed. Finally, in Section 5 we present our conclusions together with future work.

2. Concurrent Program Testing

Traditional testing techniques are often not well-suited to the testing of concurrent or parallel software, in particular when nondeterminism and concurrency features are significant. Many researchers have developed specific testing techniques addressing such issues and in addition there have been initiatives to define suitable testing criteria [16, 18, 17, 8, 11, 15]. The detection of race conditions and mechanisms for replay testing have also been investigated [4, 7, 2, 3].

Yang [18] describes a number of challenges for the testing of parallel software: 1) developing static analysis; 2) detecting unintentional races and deadlock in nondeterministic programs; 3) forcing a path to be executed when nondeterminism might exist; 4) reproducing a test execution using the same input data; 5) generating the control flow graph for nondeterministic programs; 6) providing a testing framework as a theoretical base for applying sequential testing criteria to parallel programs; 7) investigating the applicability of sequential testing criteria to parallel program testing; and 8) defining test coverage criteria based on control and data flow.

Lei and Carver [7] proposed the *reachability testing* for generating all feasible synchronization sequences (and only them). This method guarantees that every partially-ordered synchronization will be exercised exactly once without repeating any sequences that have already been exercised. The method involves the execution of the program in a semi-deterministic way; the execution is deterministic up to a given point, from which it runs nondeterministically. The resulting synchronization sequence (sync-sequence), which is feasible, is analyzed and a new feasible sequence (if possible) is computed. The authors employ a reachability schema to calculate the synchronization sequence automatically. The reachability testing uses dynamic information to execute all feasible synchronization sequences, generating all *race variants* from one particular execution.

The reachability testing process is illustrated in Figure 1 (extracted from [7]). The figure shows a space-time diagram in which vertical lines represent four threads of a concurrent program. The interaction between processes is represented by arrows from a send event to a receive event. Diagram Q_0 shows the one execution of the program, generating the synchronizations: (s_1^{T1}, r_1^{T2}), (s_2^{T4}, r_2^{T2}), (s_3^{T2}, r_3^{T3}), (s_4^{T4}, r_4^{T3}). V_1, V_2 and V_3 are *race variants* of Q_0 and feasible executions generated during reachability testing execution. A problem here is the high number of possible combination of synchronization that are generated and (for complex software) this number can be very high, restricting any practical application of the strategy. This approach has the important advantage that it will generate only feasible synchronization sequences, which is an important consideration when reducing the cost of the testing activity.

2.1. Structural Testing for Concurrent Programs

In this section we describe our test model and criteria for validation of message-passing software [14]. The test model captures control, data and communication information. The model considers that a fixed and known number of processes n is created at the initialization of the concurrent application. These processes may each execute different programs. However, each one executes its own code in its own memory space. The concurrent program is defined by a set of n parallel processes $Prog = \{p^0, p^1, \ldots p^{n-1}\}$. Each process p has its own Control Flow Graph CFG^p, that is built using the same concepts as traditional software [10]. In other words, a CFG of a process p is composed of a set of nodes N^p and a set of edges E^p. Each node n in the process p is represented by the notation n^p and corresponds to a set of commands that are sequentially executed or can be

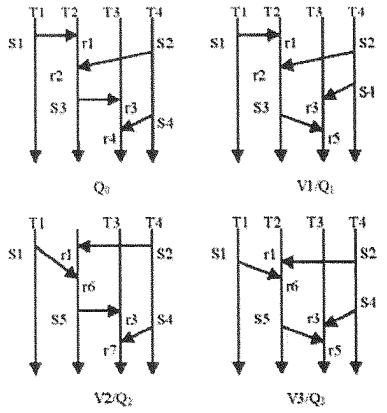


Figure 1. Example of reachability testing [7]

associated to a communication primitive (send or receive). The model considers both blocking and non-blocking receives, such that all possible interleaving between send-receive pairs can be represented. $Prog$ is associated with a Parallel Control Flow Graph ($PCFG$), which is composed of the CFG^p (for $p = 0 \ldots n-1$) and by the representation of the communication between the processes. A synchronization edge (sync-edge) (n_i^a, n_j^b) links a *send* node in a process a to a *send* node in a process b. These edges represent the possibility of communication and synchronization between processes.

A set of coverage testing criteria is defined, based on ($PCFG$): All-Nodes; All-Edges; All-Nodes-R; All-Nodes-S and All-Edges-S (related to control and synchronization information) and All-C-Uses; All-P-Uses; All-SUses; All-S-C-Uses and All-S-P-Uses (related to data and communication information) [14]. The All-Edges-S criterion requires that the test set executes paths that cover all the sync-edge associations of the concurrent program under testing; the All-S-uses criterion requires that the test set executes paths that cover all the *s-use* associations. An *s-use* is an association between a node n^p, that contains a definition of a variable x, and a sync-edge that contains a communications use of x.

An example of a $PCFG$ is shown in Figure 2. There are four processes, consisting of two different codes. Synchronization pairs are represented by dotted lines — for example, the pair $(2^0, 2^m)$ is one sync-edge between process p^0 and p^m. Each sync-edge is associated with one or more s-use associations, and related to a variable represented in $PCFG$.

Nondeterminism is the key issue addressed in this test model. As it is impossible to determine statically when a synchronization is feasible, a conservative approach is assumed, where every pair of send and receive events which have the appropriate types are considered

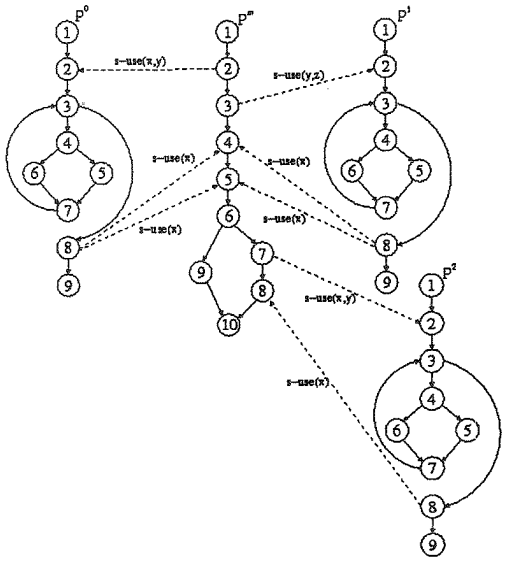


Figure 2. Example of a Parallel Control Flow Graph

as a possible matching. Considering the example of Figure 1, the required sync-edges are: (s_1^{T1}, r_1^{T2}), (s_2^{T4}, r_1^{T2}), (s_4^{T4}, r_1^{T2}), (s_1^{T1}, r_2^{T2}), (s_2^{T4}, r_2^{T2}), (s_4^{T4}, r_2^{T2}), (s_1^{T1}, r_3^{T3}), (s_2^{T4}, r_3^{T3}), (s_3^{T2}, r_3^{T3}), (s_4^{T4}, r_3^{T3}), (s_1^{T1}, r_4^{T3}), (s_2^{T4}, r_4^{T3}), (s_3^{T2}, r_4^{T3}), (s_4^{T4}, r_4^{T3}). Some sync-edges are infeasible (e.g., (s_1^{T1}, r_3^{T3})) but are required. Using controlled execution [2], it is possible to force the execution of feasible sync-edges.

It is not necessary to execute all combinations of possible synchronization as long as at least one execution of each sync-edge pair is included. A problem in this approach is the high number of infeasible sync-edges that are generated and need to be analysed. Nevertheless, it is interesting because it uses information generated statically to direct the selection of test cases and to assess the coverage of the program under test. We believe that the choice of the test case can influence the results obtained, improving the overall testing activity quality. This has led directly to the test strategy presented in the next section.

3. Proposed test strategy

In this paper we propose a test strategy that combines both reachability testing and coverage testing to execute synchronization events. The main motivation of this strategy is to improve coverage testing; however, the approach can also be applied to improve reachability testing perfor-

mance. In this case, reachability testing can be applied using an approach that selectively exercises a set of sync-sequences according to a specified coverage testing criterion. Exhaustive testing is not always practical and they pointed out the need to use mechanisms to guide the selection of the sync-sequences during reachability testing [7].

In Figure 3 the proposed test strategy is illustrated. This figure does not show all the steps necessary to apply the coverage testing criterion, only those important to our strategy.

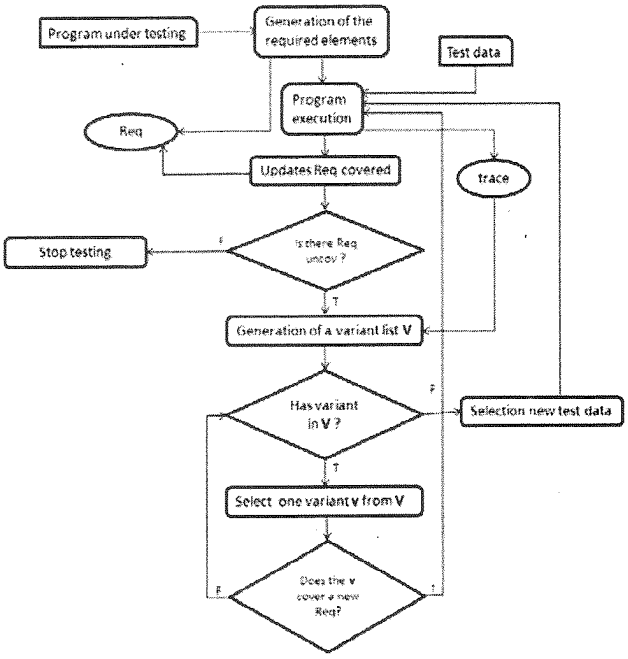


Figure 3. Test Strategy using Reachability Testing and Coverage Testing Criterion

First, a required elements list Req is generated from a given concurrent program, based on the all-s-uses and all-edges-s criteria. An initial test dataset is produced and the program is executed to generate an execution trace, containing a record of all the nodes and the sync-edges executed by the test dataset. The elements covered by the test dataset execution are marked in Req and any required element not yet covered identified. Usually, a procedure is used to select a new test dataset to improve the coverage of Req.

Considering reachability testing in this context, the next step is the generation of a list V of the variants, based on the sync-edges executed. For each sync-edge all possible variants are then generated. The difference here from reachability testing, is that only the variants required to cover a required element that is not yet covered are included. Therefore, when a new variant v is selected from V, it is verified only if it executes a new requirement of Req. Otherwise, another variant is selected or a new test dataset is generated (when V is empty). The procedure to execute a variant v is the same as that defined by Lei and Carver [7]: the controlled execution ensures that the synchronization of the v always occurs during the execution. After execution of the variant v, the execution trace is obtained and the required elements covered for this execution are marked in list Req. Considering now the variant v, new variants are generated and added to the list of the variants V. The procedure to execute variants or new test datasets is repeated while any required elements remain to be covered.

4. Experimental Study

In this section, we present an experimental study that indicates that the approach combining reachability and coverage criteria testing improve overall testing quality. The ValiMPI tool was used to conduct this study. ValiMPI is a tool developed to test concurrent programs implemented in MPI (Message Passing Interface), proposed originally to support the coverage testing mentioned in Section 2 [13, 6]. ValiMPI functionality has been extended to implement the reachability testing strategy proposed by Lei and Carver [7] and hence test the strategy proposed in this paper.

Eight different MPI programs were used in this study, implementing classical concurrency algorithms. The complexity is given by the number of sends s and receives r of each program: **sieve of Eratosthenes** - (7s and 9r) an algorithm for finding all prime numbers up to a specified integer [9]; **gcd** - (7s and 7r) to calculate the greatest common divisor of three numbers, using successive subtractions between two numbers until one of them is zero; **mmult** - (15s and 27r) to implement matrix multiplication using domain decomposition; **philosophers** - (11s and 10r) to implement the dining philosophers problem; **pairwise** - (16s and 16r) where each process n_i receives a data X_i and is responsible for computing the interactions $I(X_i, X_j)$,for $i \neq j$. For this, a structure with N channels is used, where each communication channel represents a pair source-destination — these channels are used to connect the N tasks into a unidirectional ring; **reduction** - (4s and 4r) to implement the reduction operation of distributed data, considering add, multiplication, greater than and less than operations; **qsort** - (28s and 52r) to implement quicksort, based on the parallel algorithm presented in Grama [5]; and **jacobi** - (23s and 37r) to implement Jacobi-Richardson iteration for solving a linear system of equations.

Three different test scenarios were executed:

1. **Selection of adequate test case using coverage criteria (CovT)**: using the criteria all-s-uses and all-edges-s, test cases were manually generated to exercise the required elements of these criteria, s-use associations

and sync-edges, respectively. Infeasible elements were identified to evaluate the coverage of an initial test set.

2. **Application of reachability testing (RT)**: using the initial test set generated during Scenario CovT, reachability testing was undertaken, according to the algorithm proposed by Lei and Carver [7].

3. **Application of our test strategy (RTCovT)**: using the criteria all-s-uses and all-edges-s, (related to synchronization), reachability testing was executed guided by the required elements of these criteria, following the steps discussed in Section 3.

Table 1 presents the sync-sequences generated by Reachability Testing (column RT) and by our test strategy (column RTCovT), using two different test sets: T_1, generated by CovT and containing test cases adequate to execute both sync-edges and s-uses; and T_2, which is a subset of T_1 containing only effective test cases (i.e. T_2 contains only test case contributing to the execution of the sync-edges). RT executes, for each test case, all variants of the each sync-edge, even those variants already executed previously. For this reason, the number of the sync-sequences generated by RT is higher than the sync-sequences generated by our test strategy. These results indicate that is possible to reduce the cost of the reachability testing using coverage testing. A fundamental problem with reachability testing is to decide when the testing activity can be considered complete; our test strategy contributes to the work on this problem. We compared the advantages of using our test strategy compared to the alternatives discussed previously.

Table 1. Number of the sync-sequences executed

Programs	T1	T1		T2	T2	
		RT	RTCovT		RT	RTCovT
sieve	10	60	13	4	20	7
gcd	13	24	14	7	14	9
mmult	8	48	11	4	24	5
philosophers	1	1680	2	1	1680	2
pairwise	4	4	4	1	1	1
reduction	4	4	4	1	1	1
qsort	3	794	18	3	266	16
jacobi	9	1710	13	6	377	11

Table 2 shows the results of the coverage obtained using the coverage testing (CovT) and our test strategy (RTCovT) for all-edges-s and all-s-uses criteria. For this analysis, for each program, the same test set T (generated on an *ad-hoc* basis) was used to execute the two test scenarios and the two coverage criteria. Our test strategy (RTCovT) indicates the potential to improve the criteria coverage because our strategy executes a greater number of sync-edges and s-uses than the traditional coverage testing, establishing that it is a good strategy to reduce the overall application cost of the test. Some of the programs in the test had no improvement in coverage because in these cases the T set already covered all feasible elements for the criteria.

Table 2. Coverage using coverage criteria and the test strategy

Programs	All-Edges-S		All-S-Uses	
	CovT	RTCovT	CovT	RTCovT
sieve	80.95%	90.48%	56.67%	76.67%
gcd	100.00%	100.00%	80.00%	85.00%
mmult	93.33%	93.33%	94.74%	94.74%
philosophers	100.00%	100.00%	75.00%	75.00%
pairwise	100.00%	100.00%	83.33%	83.33%
reduction	100.00%	100.00%	100.00%	100.00%
qsort	51.61%	89.25%	43.54%	67.35%
jacobi	100.00%	100.00%	84.06%	85.51%

Table 3 shows the results for the Jacobi algorithm example as different test cases ($tc1$ to $tc9$) are processed. Our testing strategy always provides better test coverage and maximal coverage is achieved after only five test cases have been considered. Table 4 provides similar results for the mmult algorithm example with test cases $tc1$ to $tc8$. For this example maximum coverage is achieved after only two of the test cases have been considered.

Table 3. Evolution of the jacobi program coverage

testcases	All-Edges-S		All-S-Uses	
	CovT	RTCovT	CovT	RTCovT
$tc1$	19.30%	19.30%	8.70%	8.70%
$tc2$	38.60%	49.12%	26.09%	34.78%
$tc3$	82.46%	94.74%	60.87%	71.01%
$tc4$	84.21%	96.49%	68.12%	78.26%
$tc5$	94.74%	100.00%	78.26%	82.61%
$tc6$	94.74%	100.00%	78.26%	82.61%
$tc7$	94.74%	100.00%	78.26%	82.61%
$tc8$	100.00%	100.00%	82.61%	84.06%
$tc9$	100.00%	100.00%	84.06%	85.51%

5. Conclusions

In this paper we have presented a new test strategy to validate concurrent programs, using a combination of coverage criteria and reachability testing. The coverage criteria are used both to select test cases and to determine the execution of new synchronizations, while the reachability testing is used to select appropriate synchronizations to be executed.

Table 4. Evolution of the mmult program coverage

testcases	All-Edges-S		All-S-Uses	
	CovT	RTCovT	CovT	RTCovT
$tc1$	60.00%	93.33%	63.16%	89.47%
$tc2$	60.00%	93.33%	68.42%	94.74%
$tc3$	73.33%	93.33%	78.95%	94.74%
$tc4$	86.67%	93.33%	89.47%	94.74%
$tc5$	86.67%	93.33%	89.47%	94.74%
$tc6$	86.67%	93.33%	89.47%	94.74%
$tc7$	86.67%	93.33%	89.47%	94.74%
$tc8$	86.67%	93.33%	94.74%	94.74%

The combination of the two approaches has the potential to deliver significant reduction in the overall testing cost. Due to the high number of synchronizations in a typical concurrent program, the execution of these synchronizations using reachability testing alone can be impractical; while in the case of the coverage criteria used by itself, these synchronizations generate a high cost because of the number of infeasible synchronizations that must be analyzed.

The test strategy described in this paper contributes in two ways: 1) by using structural criteria to minimize the number of sequences in reachability testing; and 2) by guiding the generation of test cases based on the structural criteria, using reachability testing to increase the test coverage. An experimental study has been undertaken to evaluate this approach. The results indicate that is promising to adopt this test strategy, with an improvement in test coverage in every case considered.

Finally, we plan to evaluate the proposed test strategy in terms of revealing faults. Preliminary results have demonstrated that our strategy is effective in detecting faults. The test sets discussed above were evaluated using the fault taxonomy presented in [4] and 84.8% of the seeded defects were revealed on average. Further studies are being developed using different fault taxonomies and comparing the effectiveness of our strategy with the effectiveness of reachability testing.

6 Acknowledgments

The authors would like to thank CAPES and FAPESP, Brazilian funding agencies, for the financial support, under Capes process 1191/10-1 and FAPESP processes: 2008/04614-5, 2010/02839-0.

References

[1] R. Carver and Y. Lei. Distributed reachability testing of concurrent programs. *Concurrency and Computation: Practice and Experience*, 22(18):2445–2466, 2010.

[2] R. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, pages 74–86, Mar. 1991.

[3] S. K. Damodaran-Kamal and J. M. Francioni. Nondeterminacy: Testing and debugging in message passing parallel programs. In *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 118–128, New York, May 1993.

[4] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[5] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[6] A. C. Hausen, S. R. Vergílio, S. Souza, P. Souza, and A. Simão. A tool for structural testing of MPI programs. In *8th IEEE Latin-American Test Workshop*, march 2007.

[7] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE TSE*, 32(6):382–403, June 2006.

[8] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *Proceedings of the ACM SIGSOFT symposium on the foundations of software engineering*, pages 533–536, New York, NY, USA, 2007. ACM.

[9] M. J. Quinn. *Parallel Computing : Theory and Practice*. McGraw-Hill, New York, 2nd. edition, 1994.

[10] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transaction Software Engineering*, 11(4):367–375, Apr. 1985.

[11] C. Robinson-Mallett, R. M. Hierons, J. Poore, and P. Liggesmeyer. Using communication coverage criteria and partial model generation to assist software integration testing. *Software Quality Control*, 16(2):185–211, 2008.

[12] F. S. Sarmanho, P. S. L. Souza, S. R. S. Souza, and A. S. Simão. Structural testing for semaphore-based multithread programs. In *International Conference on Computational Science, LNCS*, volume 5101, pages 337–346, 2008.

[13] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, T. G. Bliscosque, A. M. Lima, and A. C. Hausen. Valipar: A testing tool for message-passing parallel programs. In *International Conference on Software knowledge and Software Engineering (SEKE05)*, pages 386–391, Taipei-Taiwan, 2005.

[14] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, and A. C. Hausen. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience*, 20:1893–1916, mar 2008.

[15] J. Takahashi, H. Kojima, and Z. Furukawa. Coverage based testing for concurrent software. In *28th International Conference on Distributed Computing Systems Workshops, 2008.*, pages 533–538, June 2008.

[16] R. N. Taylor, D. L. Levine, and C. Kelly. Structural testing of concurrent programs. *IEEE Transaction Software Engineering*, 18(3):206–215, Mar. 1992.

[17] W. E. Wong, Y. Lei, and X. Ma. Effective generation of test sequences for structural testing of concurrent programs. In *10th IEEE International Conference on Engineering of Complex Systems (ICECCSt'05)*, pages 539– 548, 2005.

[18] C.-S. D. Yang. *Program-Based, Structural Testing of Shared Memory Parallel Programs*. PhD thesis, University of Delaware, 1999.

ISBN-10: 1-891706-29-2 (paper)
ISBN-13: 978-1-891706-29-5

Additional Copies can be ordered from:
Knowledge Systems Institute Graduate School
3420 Main Street
Skokie, IL 60076, USA
Tel:+1-847-679-3135
Fax:+1-847-679-3166
Email:office@ksi.edu
http://www.ksi.edu

Proceedings preparation, editing and printing are sponsored by
Knowledge Systems Institute Graduate School

Printed by Knowledge Systems Institute Graduate School

Apêndice

F

(Silva et al. 2012b) Silva, R. A.; Souza, S. R. S.; Souza, P. S. L. Mutation Operators for Concurrent Programs in MPI. In: 13th Latin American Test Workshop, Quito, Ecuador, 2012b, p. 69-74

Mutation Operators for Concurrent Programs in MPI

Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, Paulo Sergio Lopes de Souza
Instituto de Ciências Matemáticas e de Computação - ICMC
Universidade de São Paulo, USP
São Carlos, Brazil
{adamshuk, srocio, pssouza}@ icmc.usp.br

Abstract—**Concurrent Programming became an essential paradigm to reduce the computational time in many application domains. Mutation testing is an important criterion which uses mistakes made by software developers to derive test requirements. To apply this criterion in context of concurrent programs it is necessary to consider the implicit features of these programs, such as: communication, synchronization and non-determinism. Due to the non-determinism, special attention must be given during the mutant behavior analysis. This paper presents a set of mutation operators for concurrent programs in MPI (Message Passing Interface). This mutation operators set was defined based on typical errors of concurrent programs, extracted from literature. An example is presented to illustrate the application of the mutation operators to reveal faults in MPI programs.**

Keywords- software testing; mutation operators; concurrent programs; MPI program

I. INTRODUCTION

The testing activity of concurrent programs is more complex than the sequential ones. Features such as non-determinism, synchronization and inter-process communication increase the difficulty of validation and testing [1].

A concurrent program is composed of parallel processes that interact in order to solve a complex problem. The communication among parallel processes can be done using shared memory or message-passing paradigms. When shared memory is used, the communication occurs when a process writes in a variable that can be read and/or updated by other processes that share the same address space; the synchronization occurs through specific primitives, such as semaphores and condition variables. When message-passing is used, a process sends a message that is received by another one, using, for example, ***send()*** and ***recv()*** primitives [2]. MPI (Message Passing Interface) is a message-passing environment to support the development of concurrent applications in C or Fortran language.

For sequential programs, many testing problems were simplified with the introduction of testing criteria and the implementation of supporting tools. A testing criterion is a predicate to be satisfied by a set of test cases which can be used as a template for the generation of test data [3].

Extending previous work on sequential program testing, structural testing criteria for the validation of concurrent programs were proposed, applicable to both message-passing [4] and shared memory software [1, 5]. These testing criteria are designed to exploit information about the control, data and communication flows of concurrent programs, considering both sequential and parallel aspects.

Mutation testing is a fault-based criterion which has presented a high effectiveness to reveal faults [6]. In this criterion, faults are inserted in program under test **P** and test cases are generated to show that a program **P´**, with inserted faults, has a different behavior (probably, not correct), compared to **P**. In this sense, the objective is showing that the inserted faults are not presented in **P**, improving the quality of the test cases set.

The faults are inserted in a systematic way, using a predefined set of mutation operators. These operators are dependents of the target language and are defined to capture typical errors in this language.

In this paper it is presented a set of mutation operators for MPI programs. These operators exercise features about communication and synchronization of parallel tasks in MPI, using the message-passing paradigm. The mutation operators proposed in this paper extend the previous works [4, 7], considering in a broad sense the MPI standard and the implementation Open MPI. The definition of the mutation operators for MPI programs was motivated by the importance of this standard to develop message-passing parallel programs, which is considered the *de facto* standard for writing message-passing parallel programs [8].

This paper is organized as follows. In the Section II is described how the mutation testing criteria works. Section III presents the definition of mutation operators for MPI programs. Section IV describes the use of mutation operators in order to find an error in a program. Finally, Section V summarizes the results of our study about mutation operators for MPI programs.

II. MUTATION TESTING

Currently, the mutation testing has had great acceptance in the community of software testing, where it is considered one of the most effective criteria to reveal the presence of defects, although it is relatively expensive [9]. This criterion is based on the construction of mutants using a fault model specific to the problem domain in which the program in test is defined. To create mutants it is necessary to have a model with hypothetical faults which are usually committed during the implementation of the program. Each fault in this model becomes a mutation operator.

978-1-4673-2356-7/12$26.00 ©2012 IEEE

The fault model needs to represent plausible defects, since the mutants that are rejected by the compiler or that fail in almost all tests are not good examples of faults. A valid mutant is syntactically correct; for example, a mutant derived from an exchange between a "for" statement by an "if" statement is not valid because it results in a program with compilation error. In this way, an example of a valid mutant is the exchange between the relational operator ">" in an expression "if (a > 0)" by another relational operator, for instance, the operator "<". Thus, a mutant is useful, and valid, if the mutant behavior differs of the original program behavior for no more than a small subset of test cases [10]. For example, if a constant used in a loop statement is changed from 1 to 1000, this mutant will be valid but not useful because it adds no useful information about the effectiveness of the test suite.

The main problem associated with mutation analysis is the number of mutants generated. Often this number is too high causing a high computational cost to execute all the mutants. Furthermore, there is a problem when it is necessary identify equivalent mutants because this task is done by the tester and sometimes it takes a long time to be finished. The tester has to see the mutant code comparing with the original one in order to decide if they are equivalent or not. So, the more mutants are generated, the more mutants must be analyzed to identify equivalent mutants. Many studies have been done in the context of mutation testing. Many aspects such as theoretical and experimental studies, extensions or adaptations, definition of mutation operators, tools, etc. have been explored. Some studies aims to find a way to reduce the cost of the test, suggesting alternatives to select which mutants must be considered, for example: selective mutation [11], random mutation [12] and essential operators [13]. These strategies reduce the number of generated mutants without reduce the effectiveness to reveal defects of this criterion.

In context of concurrent programs, there are some contributions that define this testing criterion for multithreaded programs. The application of mutation testing for concurrent programs was addressed initially by [14], where the author presents an approach to the deterministic execution of concurrent programs (DET) applied to Ada language. Later, [15] presents a procedure called deterministic execution mutation testing (DEMT). References [16] and [17] propose operators for Ada language programs. Reference [18] defines a set of mutation operators to test aspects of concurrency and synchronization of Java, where the main structures related to concurrency were identified and mutation operators were defined in order to execute those structures.

In [19] there is a proposal of mutation operators for object-oriented concurrent programs in Java. The focus of the approach is concerned with the cases where the access to shared data should be protected. Reference [20] proposes the creation of mutation operators for programs developed in Java (J2SE 5.0). In [7] it is presented an initial proposal of mutation operators for programs in PVM (Parallel Virtual Machine). Reference [21] defines mutation operators for concurrent programs in SystemC, which is a modeling language used to design SoCs (System-on-Chip).

The definition of the mutation testing to concurrent programs presents some challenges. This is due to non-determinism present in concurrent programs, leading to different results that are obtained when the program is executed with the same data input. This is a problem because it is necessary to analyze the mutant behavior in relation to all possible program behavior, which is not a trivial task. In Silva-Barradas [17] it is presented a procedure to find all possible output for a test input.

III. MUTATION OPERATORS FOR MPI PROGRAMS

The MPI second version (MPI-2) specifies features, such as: parallel I/O, dynamic process management and two different message-passing communication models (one-sided and two-sided communication models). The behavior of the one-sided communication model is similar to the shared-memory paradigm used in multithreaded programs. The two-sided communication model is based on a set of primitives responsible for both the data exchanging and the processes synchronization. Senders and receivers explicitly participate of the communication, using different types of point-to-point or collective primitives. Some other examples of primitives and semantics associated to two-sided communication model are: persistent, (non-) blocking, (as) synchronous, (un) buffered and ready. These primitives and semantics are also affected by communicators that, in a practical way, determine the message reach [22]. This paper is related to two-sided communication model.

Another aspect of the motivation for the MPI mutation operators definition is the possibility of a programmer's mistake, since there are several functions with similar syntax, but with very different semantic and purpose. Besides, some functions have a considerable number of arguments. The mutation testing can be used to help the programmer to identify the different MPI functions and the possible mistakes related to them.

In general, mutation operators are designed based on the experience in the use of a given language, as well as the most common mistakes made during its use. The defects taxonomy defined by [23] was used to help the definition of mutation operators for MPI programs. The taxonomy was defined based on real mistakes committed by expert MPI programmers. Besides that, the mutation operator set proposed in this paper was defined considering the mutation operator set for PVM (Parallel Virtual Machine) [7] and the mutation operators defined for sequential programs in C, considering mutation operators for unit testing and integration testing [24, 25].

Table I shows three categories of mutation operators for MPI programs: Collective, Point-to-point and All. The categories are organized according to the function-target where the mutation operators are applied. The first category "Collective" presents mutation operators that are applied in collective-communication functions. The second category "Point-to-point" presents mutation operators that are applied in point-to-point communication functions subdivided in operators applied on send, receive or others point-to-point functions. The last category "All" presents operators that can be applied in both collective and point-to-point functions.

TABLE I. MUTATION OPERATORS FOR MPI PROGRAMS

Category		MPI Mutant Operators
Collective		ReplRoot
		ReplReduce
		ReplGather
		DelBarrier
		MoveColletiveUpDown
Point-to-point	Send	ReplModelSend
		ReplModeSend
		DelSend
	Receive	ReplSource
		ReplTag
		ReplProbe
		ReplModelRecv
		DelRecv
	Other	ReplFin
		ReplSendRecv
		ReplStart
		ReplWait
		DelFinTask
		DelDerivDType
		DelDetach
All		ReplArg
		ChanArg
		ReplComm
		ReplCall
		DelCall
		InsUnaArg

A. *Mutation Operators for Collective Function*

1) ReplRoot – Replacement of argument root: this operator replaces the value of the parameter root in some collective function calls. The parameter root specifies the main process of the communication, usually the target (destiny) to the messages. The parameter root is replaced by all processes belonging to the program. Example:

Original: MPI_Reduce (&sendbuf, &recvbuf, count, datatype, MPI_MAX, ***root***, comm);

Mutant: MPI_Reduce (&sendbuf, &recvbuf, count, datatype, MPI_MAX, ***1***, comm);

2) ReplReduce - Replacement of reduce function: this operator replaces function calls MPI_Reduce, MPI_Allreduce and MPI_Reduce_scatter. Example:

Original: ***MPI_Reduce*** (&sendbuf, &recvbuf, count, datatype, op, root, comm.);

Mutant: ***MPI_Allreduce*** (&sendbuf, &recvbuf, count, datatype, op, comm);

3) ReplGather - Replacement of probe function: this operator replaces function calls MPI_Gather, MPI_Gatherv, MPI_Allgather and MPI_Allgatherv. Example:

Original: ***MPI_Gather*** (&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm);

Mutant: ***MPI_Allgather*** (&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, comm);

4) DelBarrier - Delete barrier: this operator removes any occurrence of a call to a synchronization function MPI_Barrier.

Original: MPI_Barrier (comm);

Mutant: The function call is removed.

5) MoveColletiveUpDown - Move collective function up and down: this operator moves a collective operation to few lines up or down from the line where it is. An example is shown in the Figure 1.

B. *Mutation Operators for Point-to-point Functions*

1) ReplModelSend - Replacement of the send function: this operator replaces blocking and non-blocking send function calls, considering each one of the four possible modes of send calls. Example:

Original: ***MPI_ISend*** (&sndbuf, count, dtype, dest, tag, comm, &request);

Mutant: ***MPI_Send*** (&sndbuf, count, dtype, dest, tag, comm);

Original	Mutant

...
if(rank==0){
a=2;
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(&a,1,MPI_INT,
0,MPI_COMM_WORLD);
...
	...
	if(rank==0){
	a=2;
	MPI_Barrier(MPI_COMM_WORLD);
	}
	MPI_Bcast(&a,1,MPI_INT,0,MPI_COM
	M_WORLD);
	...

Figure 1. Original and Mutant code.

2) ReplModeSend - Replacement of the send mode: this operator replaces the mode used for communication between processes (Standard, Synchronous, Immediate and Buffered). Example:

Original: ***MPI_Send*** (&sndbuf, count, dtype, dest, tag, comm);

Mutant: ***MPI_Ssend*** (&sndbuf, count, dtype, dest, tag, comm);

3) DelSend - Delete send calls: this operator removes a function to send a message. Example:

Original: ***MPI_Send*** (&sndbuf, count, dtype, dest, tag, comm);

Mutant: The function call is removed.

4) ReplSource - Replacement of parameter source: this operator replaces the parameter MPI_ANY_SOURCE by another source used in other receive function of the program. The constant MPI_ANY_SOURCE is used in receive functions and specifies that the receive function can receive messages from any process. Example:

Original: MPI_Recv (&recvbuf, count, dtype, ***MPI_ANY_SOURCE***, tag, comm, status);

Mutant: MPI_Recv (&recvbuf, count, dtype, ***othersource***, tag, comm, status);

5) ReplTag - Replacement of parameter tag: this operator replaces the parameter MPI_ANY_TAG by another tag used in other receive functions of the program. The constant MPI_ANY_TAG is used in a receive function to specify that the function can receive messages with any tag. A *tag* is used to distinguish messages from a same sender. Example:

Original: MPI_Recv (&recvbuf, count, dtype, source, ***MPI_ANY_TAG***, comm, status);

Mutant1: MPI_Recv (&recvbuf, count, dtype, source, ***othertag***, comm, status);

6) ReplProbe - Replacement of probe function: this operator replaces the function of receiving a message by a function responsible for checking incoming messages (MPI_Probe). Example:

Original: ***MPI_Recv*** (&recvbuf, count, dtype, source, tag, comm, &status);

Mutant: ***MPI_Probe*** (source, tag, comm, &status)**;**

7) ReplModelRecv - Replacement of the receive model: this operator replaces blocking and non-blocking receive function calls. Example:

Original: ***MPI_Recv*** (&recvbuf, count, dtype, source, tag, comm, &status);

Mutant: ***MPI_Irecv*** (&recvbuf, count, dtype, source, tag, comm, &status, &request);

8) DelRecv - Delete receive calls: this operator removes a function of receiving message. Example:

Original: ***MPI_Recv*** (&recvbuf, count, dtype, source, tag, comm, &status);

Mutant: The MPI_Recv function call is removed.

9) ReplFin - Replacement of finalize functions: this operator replaces functions MPI_Finalize and MPI_Abort. Example:

Original: ***MPI_Finalize ()***;

Mutant: ***MPI_Abort (comm)*;**

10) ReplSendRecv - Replacement of the MPI_Sendrecv function: this operator replaces function MPI_Sendrecv by similar functions such as MPI_Sendrecv_replace, all modes of send, receive and probe. Example:

Original: ***MPI_Sendrecv*** (&sendbuf, sendcount, sendtype, dest, sendtag, &recvbuf, recvcount, recvtype, source, recvtag, comm, &status);

Mutant: ***MPI_Sendrecv_replace*** (&sendbuf, sendcount, sendtype, dest, sendtag, source, recvtag, comm, &status);

11) ReplStart - Replacement of start function: this operator replaces the function MPI_Start by MPI_Startall and vice-versa. Example:

Original: ***MPI Startall*** (count, array_of_requests);

Mutant: ***MPI Start*** (&request);

12) ReplWait – Replacement of wait function: this operator replaces functions MPI_Wait, MPI_Waitall, MPI_Waitany and MPI_Waitsome. Example:

Original: ***MPI Wait*** (&request, &status);

Mutant: ***MPI Waitall*** (count, array_of_requests, array_of_statuses);

13) DelFinTask - Delete function of task finalization: this operator removes a function to finalize a parallel task. The functions that can be eliminated are MPI_Finalize and MPI_Abort. Example:

Original: ***MPI_Abort (comm)***;

Mutant: The function call is removed.

14) DelDerivDType - Delete derived data type functions: this operator removes each occurrence of derived data type function. Example:

Original: ***MPI_Type_contiguous*** (count, oldtype, &newtype);

Mutant: The function call is removed.

15) DelDetach - Delete detach function: this operator removes each occurrence of a detach memory function. Example:

Original: ***MPI_Buffer_detach*** (&buffer, size);

Mutant: The function call is removed.

C. Mutation Operators for All MPI functions

1) ReplArg – Replacement of arguments in function calls: this operator replaces the parameters of each MPI function present in the program. The parameters are replaced by all variables and constants of the program (including defined constants of the MPI) and by required constants. The set of required constants, defined by [27] contains special values that are relevant to some primitive data types in C and operations associated with these types.

Example:

Original: MPI_Send (***&sndbuf***, count, dtype, dest, tag, comm);

Mutant: MPI_Send (***&count***, count, dtype, dest, tag, comm);

2) ChanArg – Change arguments in functions calls: this operator changes each argument of one MPI function presents in program for each one present in this function. Example:

Original: MPI_Send (***&sndbuf***, ***count***, dtype, dest, tag, comm);

Mutant: MPI_Send (***&count***, ***sndbuf***, dtype, dest, tag, comm);

3) ReplComm - Replacement of communicator: this operator replaces the communicator by other inter-and intra-communicator present in the program. Example:

Original: MPI_Send (&sndbuf, count, dtype, dest, tag, ***MPI_COMM_WORLD***);

Mutant: MPI_Send (&sndbuf, count, dtype, dest, tag, ***comm1***);

4) ReplCall - Replacement of function: this operator replaces a MPI function for each MPI function presents in the program. Example:

Original: ***MPI_Cancel*** (&request);

Mutant: ***MPI_Comm_size*** (comm, size)**;**

5) DelCall - Delete call: this operator removes a MPI function. Example:

Original: MPI_Start (&request);

Mutant: The function call is removed.

6) InsUnaArg - Insertion of unary operators in arguments: this operator inserts unary operators in arguments. The unary operators are: -, -1 and +1. Example:

Original: MPI_Send (&sndbuf, ***count***, dtype, dest, tag, comm);

Mutant: MPI_Send (&sndbuf, ***-count***, dtype, dest, tag, comm);

IV. APPLYING MUTATION TESTING IN AN EXAMPLE

In this section it is showed how to apply mutation operators in order to find defects in the program under test. The mutation test is used to evaluate the quality of a set of test cases. Basically, given a program under test **P** and a set of test cases **T**, whose quality is to be evaluated, the following steps can be applied: 1) generation of the mutants; 2) execution of the **P** with **T**; 3) execution of the mutants with **T**; and 4) analysis of the mutants live (mutants that have different behavior comparing with the **P** behavior) .

The Greatest Common Divisor (GCD) program is used to illustrate the application of the mutation testing for MPI programs. This program uses four processes to calculate the greatest common divisor of three numbers. The process with rank 0 runs the master process code **P0** (Fig. 2) and the others processes run the slave code (**P1, P2** and **P3**). The master process sends the first two numbers to **P1** and sends last two ones to **P2**. These two slave processes receive the values sent by **P0**, find the greatest common divisor between each pair of values, using the technique of successive subtractions, and then they send their partial-results to the **P0** (each one using a separate message). If one of the values returned to **P0** is 1, then **P3** is finalized. If these values are not 1, then the values received from **P1** and **P2** are sent to **P3** to determine the GCD of these values. The final result is returned to **P0** that print it. The code of master process is shown in Fig. 2, where the highlighted line has a fault. The fifth parameter (tag) should be 2 instead of 0 as shown.

The first step applies the mutation operators defined for MPI and generates the mutants for the GCD program. Table II shows the total of mutants generated by each mutation operator, considering only operators that generated mutants.

Considering the application of the operator "ReplArg", there is one mutant where the fifth argument of the receive function will be replaced by the constant 1, as shown in Fig. 3.

```
void Master(int x, int y, int z)
{
   int buf[3];

   buf[0] = x;
   buf[1] = y;
   MPI_Send(buf, 2, MPI_INT, 1,
1, MPI_COMM_WORLD);

   buf[0] = y;
   buf[1] = z;
   MPI_Send(buf, 2, MPI_INT, 2,
1, MPI_COMM_WORLD);

   MPI_Recv(buf, 1, MPI_INT,
MPI_ANY_SOURCE, 2,
MPI_COMM_WORLD, &status);
   x = buf[0];

   MPI_Recv(buf, 1, MPI_INT,
MPI_ANY_SOURCE, 2,
MPI_COMM_WORLD, &status);
   y = buf[0];

   if (x > 1 && y > 1)
   {
      buf[0] = x;
      buf[1] = y;
      MPI_Send(buf, 2, MPI_INT,
3, 1, MPI_COMM_WORLD);
→ MPI_Recv(buf, 1, MPI_INT,
MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD, &status);
      z = buf[0];
   }
   else
   {
      buf[0] = -1;
      buf[1] = -1;
      MPI_Send(buf, 2, MPI_INT,
3, 1, MPI_COMM_WORLD);
      MPI_Recv(buf, 2, MPI_INT,
3, 1, MPI_COMM_WORLD,
&status);
      z = 1;
   }
   printf("result = %d\n", z);
}
```

Figure 2. Code of master process of GCD.

TABLE II. NUMBER OF MUTANTS GENERATED

MPI Mutation Operators	Mutants Total
ReplModelSend	5
ReplModeSend	15
DelSend	5
ReplSource	5
ReplTag	5
ReplProbe	5
ReplModelRecv	5
DelRecv	5
ReplFin	1
DelFinTask	1
ReplArg	13938
ChanArg	121
ReplComm	121
ReplCall	169
DelCall	13
InsUnaArg	207
Total	14621

```
MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, &status);
```

Figure 3. Mutant code.

TABLE III. INITIAL TEST CASES

Test Input	Synchronization Sequence	Expected Output
25 26 28	SyncSeq	1
13 15 30	SyncSeq	1
47 54 73	SyncSeq	1

The second step executes test cases with the original program. Applying the test cases of the Table III in the original

program does not reveal any error in the program. Thus, it would be possible to think that the program is correct, but it is known that the highlighted line in the code has a fault and, for this reason, the application of the criteria should be done in order to improve the test cases set.

The third step applies the test cases in all mutants in order to demonstrate that the mutants have a different behavior compared to the original program. This occurs when the output of the mutant is different from the expected. After applied the set of test cases in all mutants, the mutants that are still alive must be analyzed. The mutant shown in Fig. 3 belongs to the group of mutants that remained alive.

To kill this mutant it is necessary to create a test case that exercises the point where the mutation occurs and makes the program to produce a different output than expected. The test case shown in Table IV is inserted into the test case set to do this. The original program is executed with this new test case and an unexpected output is observed. Thus, an error is found in the program and it should be corrected. After that, the test criterion is applied again in the fixed program.

TABLE IV. NEW TEST CASE

Test Input	Synchronization Sequence	Expected Output
33 42 51	SeqSync	3

V. CONCLUSION

This paper introduces and categorizes new mutation operators for MPI programs. The definition of the mutation operators for MPI programs was motivated by the importance of this standard to develop message-passing parallel programs, which is considered the *de facto* standard for writing parallel programs. The mutation operators try to represent typical defects present in this application. The operators proposed in this paper aggregate intrinsic features of the MPI standard, such as point-to-point and collective communication primitives. This work is in development and the next step will be to implement a support tool and to develop experiments to evaluate the applicability and effectiveness of the mutation operators defined.

ACKNOWLEDGMENT

The authors would like to thanks FAPESP, Brazilian funding agency, for the financial support, under process 2010/04935-6.

REFERENCES

[1] CSD. Yang. Program-based, structural testing of shared memory parallel programs. PhD Thesis, University of Delaware, 1999.

[2] A. S. Tanenbaum, A. S. Woodhull, Operating Systems Design and Implementation, 3rd ed, Prentice Hall, 2006.

[3] S. Rapps, and E.J. Weyuker. Selecting software test data using data flow information. IEEE TSE, 11(4):367-375, April. 1994.

[4] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, and A. C. Hausen. Structural testing criteria for message-passing parallel programs. Concurrency and Computation: Practice and Experience, 20:1893-1916, March 2008.

[5] F. S. Sarmanho, P. S. L. Souza, S. R. S. Souza, and A. S. Simão. Structural testing for semaphore-based multithread programs. In International Conference on Computational Science, p. 337-346, 2008.

[6] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. Software - Practice and Experience, 26(2):165–176, February 1996.

[7] C. Giacometti, S. R. S. Souza, and P. S. L. Souza. Mutation Testing for Validation of PVM Applications. In REIC. Revista Eletrônica de Iniciação científica, volume 2, 2003. (in Portuguese)

[8] B. J. Choi, A. P. Mathur, and A. P. Pattison. Pmotha: Scheduling mutants for execution on a hypercube. In: 3rd Symposium on Software Testing, Analysis and Verification, p. 58-65, Key West, FL, December 1989.

[9] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza. Concurrent Program Testing. In: M. E. Delamaro, J. C. Maldonado, and M. Jino, eds. Introduction to the Software Testing, v. 1, Elsevier Editora Ltda, 2007. (in Portuguese)

[10] M. Pezzè, and M. Young. Software testing and analysis: process, principles, and techniques. John Wiley & Sons, 2005.

[11] A. J. Offutt, G. Rothermel, C. Zapf. An experimental evaluation of selective mutation. In: Proceedings of the 15th international conference on Software Engineering, ICSE '93, Los Alamitos, CA, USA, 1993, p. 100-107.

[12] A. T. Acree. On Mutation. PhD thesis, Georgia Institute of Technology, Atlanta, GA, EUA, August 1980.

[13] E. F. Barbosa, A. M. R. Vincenzi, J. C. Maldonado. Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas C. In: XII Simpósio Brasileiro de Engenharia de Software (SBES 98), Maringá, BR, 1998, p. 103-120. (in Portuguese)

[14] K. C. Tai. On testing concurrent programs. In: COMPSAC, p. 510-517, 1985.

[15] R. Carver. Mutation-based testing of concurrent programs. In Test Conference. Proceedings. International. pages 845-853, 1993.

[16] A. J. Offutt, J. Voas, and J. Payne. Mutation operators for Ada. Technical report, 1996.

[17] S. Silva-Barradas. Mutation analysis of concurrent software. PhD dissertation, Dottorato di Ricerca in Ingegneria Informatica e Automatica, Politecnico di Milano, 1998.

[18] M. Delamaro, J. Maldonado, M. Pezzè, and A. Vincenzi. Mutant operators for testing concurrent Java programs. In XV SBES, 2001.

[19] S. Ghosh. Towards measurement of testability of concurrent object-oriented programs using fault insertion: a preliminary investigation. In Second IEEE International Workshop on Source Code Analysis and Manipulation., pages 17-25, 2002.

[20] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent Java (J2SE 5.0). In Proc. II Workshop on Mutation Analysis, pages 11, Washington, DC, USA. IEEE Computer Society, 2006.

[21] A. Sen. Mutation operators for concurrent systemc designs. In 10th International Workshop on Microprocessor Test and Verification, MTV '09, p. 27-31, Washington, DC, USA. IEEE Computer Society, 2009.

[22] M. P. I. Forum. MPI: A Message-Passing Interface Standard Version 2.2, 2009.

[23] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with intel message checker. In 2nd international workshop on Software engineering for high performance computing system applications, SE-HPCS '05, pages 78-82, 2005.

[24] H. Agrawal, R. A. DeMilo, R. Hathaway, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, EUA, March 1989.

[25] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro. Unit and Integration Testing for C Programs Using Mutation-based Criteria. Software Testing, Verification & Reliability, v. 11, n. 4, p. 249-268, 2001.

Apêndice

G

(Sarmanho et al. 2008) Sarmanho, F. S.; Souza, P. S. L.; Souza, S. R. S.; Simao, A. S. Structural testing for semaphore-based multithread programs. In: ICCS - International Conference on Computational Science, Lecture Notes in Computer Science, 5101, Krakow: Springer-Verlag, 2008, p. 337-346.

Structural Testing for Semaphore-Based Multithread Programs*

Felipe S. Sarmanho, Paulo S.L. Souza,
Simone R.S. Souza, and Adenilso S. Simão

Universidade de São Paulo, ICMC, São Carlos - SP, 668, Brazil
{sarmanho,pssouza,srocio,adenilso}@icmc.usp.br

Abstract. This paper presents structural testing criteria for validation of semaphore-based multithread programs exploring control, data, communication and synchronization information. A post-mortem method based on timestamps is defined to determine the implicit communication among threads using shared variables. The applicability of the coverage testing criteria is illustrated by a case study.

Keywords: software testing, multithread programs, testing criteria.

1 Introduction

Concurrent programming is important to reduce the execution time in several application domains, such as image processing and simulations. A concurrent program is a group of processes (or threads) that execute simultaneously and work together to perform a task. These threads access a common addressing space and interact through memory (using shared variables). The most common method to develop multithread programs is to use thread libraries, like PThreads (POSIX Threads).

Concurrent program testing is not trivial. Features like synchronization, inter-thread communication and non-determinism make this activity complex [1]. Multiple executions of a concurrent program with the same input may present different results due to different synchronization and communication sequences. Petascale systems also add more factors to this scenario, making it even worse [2].

Structural testing is a test technique that use source code information to guide the testing activity. Coverage criteria are defined to apply structural testing. A testing criterion is a predicate to be satisfied by a set of test cases and can be used as a guide for the test data generation. It is also a good heuristic to indicate defects on programs and thus to improve their quality. This activity is composed of: (1) *static analysis* to obtain the necessary data about the source code, and usually obtaining a Control Flow Graph (CFP) [3]; (2) *determining required elements* for the coverage criterion chosen; and (3) *analyzing the coverage* reached in source code by test cases, based on coverage criterion.

* This work is supported by CNPq.

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 337–346, 2008.

In the literature there are some works that address testing of concurrent programs [4, 5, 6, 7, 8, 9]. Most of these works propose a test model to represent the concurrent program and to support the testing application.

The Concurrency States Graph (*CG*) is a CFG extension proposed in [4], in which nodes represent concurrency states while the edges represent the actions required for transition between these states. That work considers concurrent languages with explicit synchronization using rendezvous-style mechanisms, such as Ada and CSP. It presents coverage criteria adapted for the *CG*; however, its usage is limited in the practice by the state space explosion problem.

PPFG (Parallel Program Flow Graph) is a graph where was inserted the concept of synchronization node to the *CFG* [5, 10]. In the *PPFG* each process that composes the program has its own *CFG*. The synchronization nodes are then connected based on possible synchronizations. This model was proposed to adapt the all-du-path criterion to concurrent programs.

PCFG (Parallel Control Flow Graph) also adapts the *CFG* for the context of parallel programs in message passing environments [7]. The *PCFG* includes the concept of synchronization nodes that are used to represent the send and receive primitives. The concept of variables was extended, to consider the concept of communicational use (s-use). Coverage criteria were also proposed in [7], based on models of control and data flow for message passing programs.

Lei and Carver propose an approach to reachability testing. Reachability testing is a combination of deterministic and non-deterministic execution, where the information and the required elements are generated on-the-fly, without static analysis [6]. This proposal guarantees all feasible synchronization sequences will be exercised at least once. The lack of static analysis means it cannot say how many executions are required. This causes the state space explosion problem. In recent works, Lei et. al [9] presents a combinatorial approach, called t-way, to reduce the number of synchronization sequences to be executed.

These related works bring relevant improvements for concurrent program testing. However, few works are found that investigate the application of the testing coverage criteria and supporting tools in the context of multithreading programs. For these programs, new aspects need to be considered. For instance, data flow information must consider that an association between one variable definition and its use can occur in different threads. The implicit inter-thread communication that occurs through shared memory makes complex the test activity. The investigation of these challenges it is not a trivial task and presents many difficulties. To overcome these difficulties, we present a family of structural testing criteria for semaphore-based multithread programs and a new test model for the support to the criteria. This model includes important features, such as: synchronization, communication, parallelism and concurrency. These data are collected using static and dynamic analyses. Information about communication is obtained after the execution of an instrumented version of the program, using a post-mortem methodology. This methodology has been adapted from Lei and Carver work [6]. Testing criteria were defined to exploit the control and data flows of these programs, considering their sequential and parallel aspects. The

main contribution of the testing criteria proposed in this paper is to provide a coverage measure that can be used for evaluating the progress of the testing activity. This is important to evaluate the quality of test cases, as well as, to consider that a program has been tested enough. It is important to point out that the objective this work is not to debug concurrent programs which already have an error revealed.

2 Model Test for Shared Memory Programs

Let $MT = \{t^0, t^1, ..., t^{n-1}\}$ be a multithread program composed of n threads denoted by t^i. Threads can execute different functionalities but all they share the same memory address space. They may also use an additional private memory. Each thread t has its own control flow graph CFG^t that is built by using the same concepts of traditional programs [3]. In short, a CFG of a thread t is composed of a set of nodes N^t and a set of edges E_I^t. These edges that link nodes in the same thread are called intra-thread edges. Each node n in the thread t is represented by the notation n_i^t, a well-known terminology in the software testing context. Each node corresponds to a set of commands that are sequentially executed or can be associated to a synchronization primitive (*post* or *wait*).

A multithread program MT is associated with a Parallel Control Flow Graph for Shared Memory ($PCFG_{SM}$), which is composed of both the CFG^t (for $0 \leq t < n$) and the representation of the synchronization among threads. N and E represent the set of nodes and edges of the $PCFG_{SM}$, respectively. For construction of the $PCFG_{SM}$, it is assumed that (1) n is fixed and known at compilation time; (2) there is implicit communication by means of shared variables; (3) there is explicit synchronization using semaphores (which has two basic atomic primitives: *post* (or p) and *wait* (or w)); and (4) initialization and finalization of threads act as a synchronization over a virtual semaphore.

Three subsets of N are defined: N_t (nodes in the thread t), N_p (nodes with *post* primitives) and N_w (nodes with *wait* primitives). For each $n_i^t \in N_p$, a set $M_w(n_i^t)$ is associated, such that for each $n_i^t \in N_p$, with a *post* to a semaphore *sem*, we define $M_w(n_i^t)$ as the set of nodes $n_j^q \in N_w$, such that exist a thread $q \in [0..n-1]$ and a *wait* primitive with respect to (w.r.t.) *sem* in n_j^q. In a similar way, for each $n_i^t \in N_w$, a set $M_p(n_i^t)$ is associated, such that for each $n_i^t \in N_w$, with a *wait* to a semaphore *sem*, we define $M_p(n_i^t)$ as the set of nodes $n_j^q \in N_p$, such that exist a thread $q \in [0..n-1]$ and a *post* primitive w.r.t. *sem* in n_j^q. In other words, $M_w(n_i^t)$ contains all possible *wait* nodes that can match with n_i^t and $M_p(n_i^t)$ contains all possible *post* nodes that can match with n_i^t.

Using the above definitions, we also define the set $E_S \subset E$ that contains edges that represent the synchronization (edge-s) between two threads, such that:

$$E_S = \{(n_j^t, n_k^q) \mid n_j^t \in M_p(n_k^q) \wedge n_k^q \in M_w(n_j^t)\} \quad (1)$$

The concurrent program shown in the Fig. 1 is used to illustrate these definitions. This program implements the producer-consumer problem with limited buffer, using PThreads library in ANSI C. There are three threads: (1) a master,

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t  mutex, empty, full;
int queue[2], avail;

void *producer(void);
void *consumer(void);

/*Thread T0*/
int main(void) {
    pthread_t prod_h, cons_h;

    avail = 0;                          /*1*/
    sem_init (&mutex, 0 , 1);           /*2,3*/
    sem_init(&empty, 0, 2);             /*4,5,6*/
    sem_init(&full, 0, 0);              /*7*/
    pthread_create(&prod_h, 0,
                   producer, NULL);/*8*/
    pthread_create(&cons_h, 0,
                   consumer, NULL);/*9*/
    pthread_join(prod_h, 0);            /*10*/
    pthread_join(cons_h, 0);            /*11*/
    exit(0);                            /*12*/
}
```

```c
/*Thread T1*/
void *producer(void) {              /*1*/
    int prod = 0, item;             /*2*/
    while (prod < 2) {              /*3*/
        item = rand()%1000;         /*4*/
        sem_wait(&empty);           /*5*/
        sem_wait(&mutex);           /*6*/
        queue[avail] = item;        /*7*/
        avail++;                    /*8*/
        prod++;                     /*9*/
        sem_post(&mutex);           /*10*/
        sem_post(&full);            /*11*/
    }
    pthread_exit(0);                /*12*/
}
/*Thread T2*/
void *consumer(void) {              /*1*/
    int cons = 0, my_item;          /*2*/
    while (cons < 2) {              /*3*/
        sem_wait(&full);            /*4*/
        sem_wait(&mutex);           /*5*/
        cons++;                     /*6*/
        avail--;                    /*7*/
        my_item = queue[avail];     /*8*/
        sem_post(&mutex);           /*9*/
        sem_post(&empty);           /*10*/
        printf("consumed: %d\n",
                        my_item);   /*11*/
    }
    pthread_exit(0);                /*12*/
}
```

Fig. 1. Producer-Consumer implemented with PThreads/ANSI C

which initializes variables and creates the producer and consumer threads; (2) a producer, which populates the buffer; (3) a consumer, which removes items from the buffer for further processing. Table 1 contains values of all sets introduced above.

Figure 2 shows the $PCFG_{SM}$ for the program in the Fig. 1. t^0, t^1 and t^2 represent the master, producer and consumer threads, respectively. Dotted lines represent synchronization edges. Some examples of synchronization edges are: $(9^2, 6^1)$ is a synchronization over semaphore, $(9^0, 1^2)$ is a synchronization of initialization and $(12^1, 10^0)$ is a synchronization of finalization. Note that there may exist internal synchronization edges, such as $(10^1, 6^1)$ and $(9^2, 5^2)$ in Fig. 2.

A path $\pi^t = (n_1^t, n_2^t, ..., n_j^t)$, where $(n_i^t, n_{i+1}^t) \in E_I^t$, is intra-thread if it has no synchronization edges. A path that includes at least one synchronization edge is called an inter-thread path and is denoted by $\Pi = (PATHS, SYNC)$, where $PATHS = \{\pi^1, \pi^2, ..., \pi^n\}$ and $SYNC = \{(p_i^t, w_j^q) \mid (p_i^t, w_j^q) \in E_S\}$ [7]. Here p_i^t is a *post* node i in thread t and w_j^q is a *wait* node j in thread q.

$PCFG_{SM}$ also captures information about data flow. Besides local variables, multithread programs have more two special types of variables: (1) shared variables, used for communication; and (2) synchronization variables, used by semaphores. V denotes all variables. $V_L^t \subset V$ contains local variables of thread t. $V_C \subset V$ contains the shared variables and $V_S \subset V$ contains the synchronization variables. Therefore, we define: $def(n_i^t) = \{x \mid x \text{ is a variable defined in } n_i^t\}$.

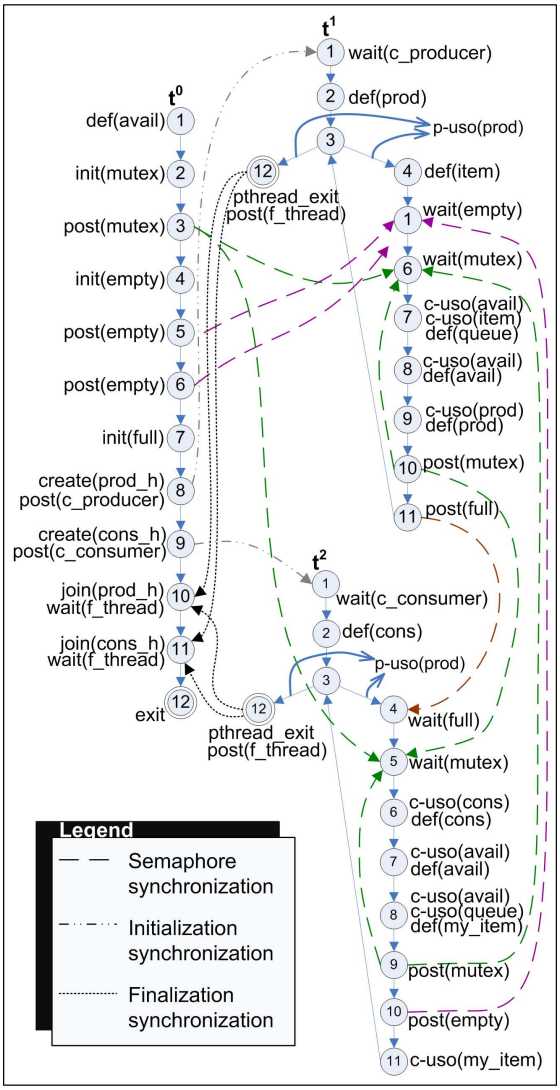


Fig. 2. PCFG_{SM} graph that represents the program shown in Fig. 1

Table 1. Sets of the test model introduced for the program shown in Fig. 1

$n = 3$	
$MT = \{t^0, t^1, t^2\}$	
$N_0 = \{1^0, 2^0, 3^0, ..., 12^0\}$	
$N_1 = \{1^1, 2^1, 3^1, ..., 12^1\}$	
$N_2 = \{1^2, 2^2, 3^2, ..., 12^2\}$	
$N = N_0 \cup N_1 \cup N_2$	
$N_p = \{3^0, 5^0, 6^0, 8^0, 9^0, 10^1, 11^1, 12^1, 9^2, 10^2, 12^2\}$	
$N_w = \{10^0, 11^0, 1^1, 5^1, 6^1, 1^2, 4^2, 5^2\}$	
$E_I^0 = \{(1^0, 2^0), (2^0, 3^0), ..., (10^0, 11^0), (11^0, 12^0)\}$	
$E_I^1 = \{(1^1, 2^1), (2^1, 3^1), ..., (11^1, 3^1), (3^1, 12^1)\}$	
$E_I^2 = \{(1^2, 2^2), (2^2, 3^2), ..., (11^2, 3^2), (3^2, 12^2)\}$	
$E_s = \{(3^0, 6^1), (3^0, 5^2), (5^0, 5^1), (6^0, 5^1), (8^0, 1^1),$	
$(9^0, 1^2), (10^1, 5^2), (10^1, 6^1), (11^1, 4^2), (12^1, 10^0),$	
$(12^1, 11^0), (9^2, 6^1), (9^2, 5^2), (10^2, 5^1), (12^2, 10^0),$	
$(12^2, 11^0)\}$	
$E = E_I^0 \cup E_I^1 \cup E_I^2 \cup E_s$	
$M_w(3^0) = \{6^1, 5^2\}$	$M_p(10^0) = \{12^1, 12^2\}$
$M_w(5^0) = \{5^1\}$	$M_p(11^0) = \{12^1, 12^2\}$
$M_w(6^0) = \{5^1\}$	$M_p(1^1) = \{8^0\}$
$M_w(8^0) = \{1^1\}$	$M_p(5^1) = \{5^0, 6^0, 10^2\}$
$M_w(9^0) = \{1^2\}$	$M_p(6^1) = \{3^0, 10^1, 9^2\}$
$M_w(10^1) = \{6^1, 5^2\}$	$M_p(1^2) = \{9^0\}$
$M_w(11^1) = \{4^2\}$	$M_p(4^2) = \{11^1\}$
$M_w(12^1) = \{10^0, 11^0\}$	$M_p(5^2) = \{3^0, 10^1, 9^2\}$
$M_w(9^2) = \{5^2, 6^1\}$	
$M_w(10^2) = \{5^1\}$	
$M_w(12^2) = \{10^0, 11^0\}$	
$V_L^0 = \{prod\_h, cons\_h\}$	
$V_L^1 = \{prod, item\}$	
$V_L^2 = \{cons, my\_item\}$	
$V_C = \{queue, avail\}$	
$V_S = \{mutex, full, empty\}$	
$def(1^0) = \{avail\}$	$def(9^1) = \{prod\}$
$def(2^1) = \{prod\}$	$def(2^2) = \{cons\}$
$def(4^1) = \{item\}$	$def(6^2) = \{cons\}$
$def(7^1) = \{queue\}$	$def(7^2) = \{avail\}$
$def(8^1) = \{avail\}$	$def(8^2) = \{my\_item\}$

A path $\pi^t = (n_1, n_2, ..., n_j, n_k)$ is definition-clear w.r.t. a local variable $c \in V_L^t$ from n_1 to node n_k or edge (n_j, n_k), if $x \in def(n_1)$ and $x \notin def(n_i)$, for $i \in [2..j]$. The notion definition-clear path is not applicable to shared variables because the communication (definition and use of shared variables) in threads is implicit. It is hard to establish a path that statically defines and uses shared variables. In Secion 2.1, we present a method to determine *execution-based* definition-clear paths for shared variables using a post-mortem methodology.

The use of variables in multithread programs can be: ***computational use (c-use)***: computational statements related to local variable $x \in V_L^t$; ***predicative use (p-use)***: conditional statements that modify the control flow of the thread and are related to local variable $x \in V_L^t$; ***synchronization use (sync-use)***: synchronization statements on semaphores-variable $x \in V_S$;***communicational c-use (comm-c-use)***: computational statements related to shared variable

$x \in V_C$; and ***communicational p-use (comm-p-use)***: conditional statements used on control flow of the thread, related to shared variable $x \in V_C$.

Based on these definitions, we establish associations between variable definition and use. Five kinds of associations are defined: ***c-use association***: is defined by a triple (n_i^t, n_j^t, x) iff $x \in V_L^t$, $x \in def(n_i^t)$, n_j^t has a c-use of x and there is at least one definition-clear path w.r.t. x from n_i^t to n_j^t. ***p-use association***: is defined by a triple $(n_i^t, (n_j^t, n_k^t), x)$ iff $x \in V_L^t$, $x \in def(n_i^t)$, (n_j^t, n_k^t) has a p-use of x and there is at least one definition-clear path w.r.t. x from n_i^t to (n_j^t, n_k^t). ***sync-use association***: is defined by a triple $(n_i^t, (n_j^t, n_k^q), sem)$ iff $sem \in V_S$, (n_j^t, n_k^q) has a sync-use of sem and there is at least one definition-clear path w.r.t. sem from n_i^t to (n_j^t, n_k^q). ***comm-c-use association***: is defined by a triple (n_i^t, n_j^q, x) iff $x \in V_C$, $x \in def(n_i^t)$ and n_j^q has a c-use of the shared variable x. ***comm-p-use association***: is defined by a triple $(n_i^t, (n_j^q, n_k^q), x)$ iff $x \in V_C$, $x \in def(n_i^t)$ and (n_j^q, n_k^q) has a p-use of the shared variable x.

2.1 Applying Timestamps to Determine Implicit Communication

In this section, we present a method to establish pairs of definition and use of shared variables. These pairs are obtained after execution of the multithread program identifying the order that the concurrent events happened. Lamport [11] presented a way to order concurrent events by means of a happens-before relationship. This relationship can determine if an event e_1 occurs before an event e_2, denoted by $e_1 \prec e_2$.

To obtain this happens-before relationship it is necessary to assign timestamps to concurrent events. Lie and Carver [6] presented a method to assign timestamps that use local logical clock. We adapt this method to assign timestamps in our testing method. The method obtains all synchronizations that happened for an execution and thus generates the communication events.

The method assigns a local logical clock vector, denoted by $t^i.cv$, for each thread t^i. This vector has dimension n, where n is the total number of threads. Each position $i \in [0..n-1]$ on the clock vector is associated to thread t^i. The clock-vector position i is updated when a new event occurs in thread t^i. For instance, observe the c_1 event in t^0 (before the clock vector was $[0,0,0]$). When a synchronization event occurs in t^j other i positions, for $i \neq j$, of the clock-vector can also be updated. For instance, considering the match (p_2, w_2). Before this synchronization the clock vector associated with t_2 was $[0,0,0]$. After, the values were updated to $[2,4,1]$.

The logical space-time diagram shown in the Fig. 3 illustrates the method, using a hypothetical example. This diagram only considers events of synchronization (p_i and w_j) and communication (c_k). Vertical lines represent the logical time of each thread. Arrows among threads represent synchronization events matching *post* and *wait* events. For instance, wait events w_1 and w_2 race the same post event p_1, but the match (w_1, p_1) has occurred.

It is possible that a *wait* primitive has several *posts* to match. These *posts* are inserted in a queue. Our method considers the access criterion *LIFO* to get the happens-before relationship. We chose *LIFO* to get most updated timestamps.

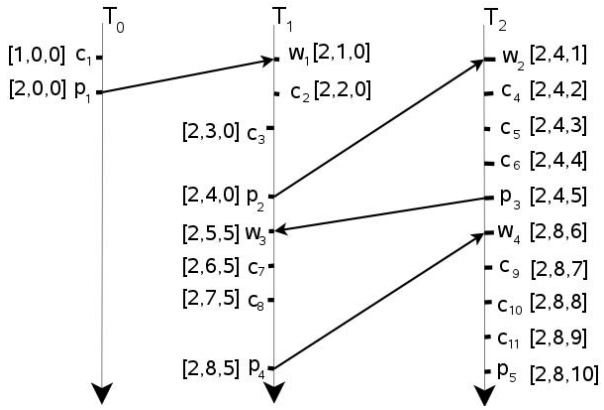


Fig. 3. Example of logical space-time diagram

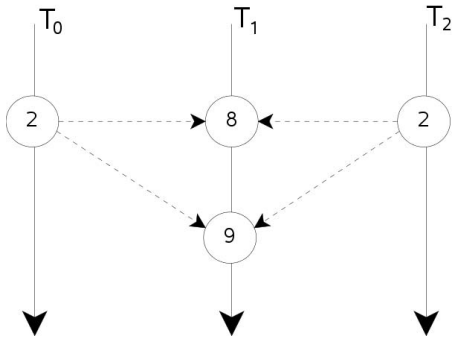


Fig. 4. Example of nondeterminism

Rules defined in [6] are used to establish if an event e_1 happens-before an event e_2. These rules are not showed here for sake of space. With this method, it is possible to show the communications that happened for a program execution.

3 Coverage Criteria

Based on the control, data and communication flow models and definitions presented in previous section, we propose two sets of structural testing criteria for shared-memory parallel programs. These criteria allow the testing of sequential and parallel aspects of the programs.

Control Flow and Synchronization-based Criteria

All-p-nodes criterion: the test set must execute all nodes $n_i^t \in N_p$.
All-w-nodes criterion: the test set must execute all nodes $n_i^t \in N_w$.
All-nodes criterion: the test set must execute all nodes $n_i^t \in N$.
All-s-edges criterion: the test set must execute all sync edges $(n_i^t, n_j^q) \in E_s$.
All-edges criterion: the test set must execute all edges $(n_i, n_j) \in E$.

Data Flow and Communication-based Criteria

All-def-comm criterion: the test set must execute paths that cover an association comm-c-use or comm-p-use for all definition of $x \in V_c$.
All-def criterion: the test set must execute paths that cover an association c-use, p-use, comm-c-use or comm-p-use for all definition of $x \in def(n_i^t)$.
All-comm-c-use criterion: the test set must execute paths that cover all comm-c-use associations.
All-comm-p-use criterion: the test set must execute paths that cover all comm-p-use associations.
All-c-use criterion: the test set must execute paths that cover all c-use associations.
All-p-use criterion: the test set must execute paths that cover all p-use associations.

All-sync-use criterion: the test set must execute paths that cover all sync-use associations.

It is necessary to know which path was exercised to evaluate the required elements covered from an execution. One option to obtain this information is to instrument the source code to produce execution trace. This instrumentation can change the original program behaviour. However, this interference does not affect the structural testing proposed here, because it does not prevent the extraction and the future execution of all possible pairs of synchronization.

Due to non-determinism, executions of a program with the same input can cause different event sequences to occur. The Fig. 4 shows the example where the nodes 8^1 and 9^1 in t^1 have non-deterministic *waits* and in nodes 2^0 (t^0) and 2^2 (t^2) have *post* to t^1. All these operations are on the same semaphore. This case illustrates the possible synchronizations among these threads. During the testing activity is essential to guarantee that these synchronizations are executed. Controlled execution is a mechanism used to achieve deterministic execution, i.e. two executions of the program with the same input are guaranteed to execute the same instruction and the specified synchronization sequence. The controlled execution used in this work was adapted from Carver method [12].

The Table 2 shows some required elements for the criteria defined in this section. These required elements are taken on the static analysis.

Table 2. Some required elements by the proposed criteria for the program of the Fig. 1

Criteria	Required Elements	Total
All-nodes-p	$3^0, 5^0, 6^0, 8^0, 9^0, 10^1, 11^1, 12^1, 9^2, 10^2, 12^2$	11
All-nodes-w	$10^0, 11^0, 1^1, 5^1, 6^1, 1^2, 4^2, 5^2$	8
All-nodes	$1^0, 2^0, 3^0, ..., 12^0, 1^1, 2^1, ..., 12^1, 1^2, 2^2, ..., 12^2$	36
All-edges-s	$(3^0, 6^1), (3^0, 5^2), (5^0, 5^1), (6^0, 5^1), (8^0, 1^1), (9^0, 1^2),$ $(10^1, 5^2), (10^1, 6^1), (11^1, 4^2), (12^1, 10^0), (12^1, 11^0), (9^2, 6^1), ...$	16
All-edges	$(1^0, 2^0), (2^0, 3^0), ..., (11^0, 12^0), (1^1, 2^1), (2^1, 3^1), ..., (11^1, 3^1), (3^1, 12^1),$ $(1^2, 2^2), ..., (11^2, 3^2), (3^2, 12^2), (3^0, 6^1), (3^0, 5^2), ..., (9^0, 1^2), (10^1, 6^1), ...$	51
All-def-comm	$(1^0, 7^1, avail), (8^1, 7^1, avail), (7^2, 8^2, avail), (7^1, 8^2, queue)$	4
All-def	$(1^0, 8^2, avail), (2^1, (3^1, 4^1), prod), (4^1, 7^1, item), (7^1, 8^2, queue),$ $(8^1, 7^1, avail), (6^2, (3^2, 12^2), cons), ...$	10
All-comm-c-use	$(1^0, 7^1, avail), (1^0, 7^2, avail), (7^2, 7^2, avail),$ $(7^2, 8^1, avail), (7^1, 8^2, queue), ...$	13
All-comm-p-use	$\emptyset$	0
All-c-use	$(8^1, 7^2, avail), (7^2, 8^2, avail), (2^1, 9^1, prod), (9^1, 9^1, prod),$ $(4^1, 7^1, item), (7^1, 8^2, queue), (8^2, 11^2, my_i item)...$	16
All-p-use	$(2^1, (3^1, 4^1), prod), (2^1, (3^1, 12^1), prod), (6^2, (3^2, 4^2), cons),$ $(6^2, (3^2, 12^2), cons), ...$	8
All-sync-use	$(2^0, (3^0, 6^1), mutex), (2^0, (3^0, 5^1), mutex), (5^0, (6^0, 5^1), empty),$ $(6^1, (10^1, 6^1), mutex), (5^2, (9^2, 6^1), mutex), ...$	14

4 Case Study

In order to illustrate the proposed testing criteria, consider the program in Fig. 1. The buffer is limited to two produced/consumed items. Due to threads scheduling, two executions are possible: (1) produce, consume, produce, consume

(*PCPC*); (2) produce, produce, consume, consume (*PPCC*). Using controlled execution, it is possible to force the order these executions. Considering to first execution (PCPC) the executed paths and their synchronizations are:

π^0 = {1,2,3,4,5,6,8,9,10,11,12}
π^1 = {1,2,3,4,5,6,8,9,10,11,3,4,5,6,7,8,9,10,11,12}
π^2 = {1,2,3,4,5,6,8,9,10,11,3,4,5,6,7,8,9,10,11,12}
$SYNC$= $\{(8^0, 1^1), (6^0, 5^1), (3^0, 6^1), (9^0, 1^2), (11^1, 4^2), (10^1, 5^2),$
$(5^0, 5^1), (9^2, 6^1), (12^1, 10^0), (11^1, 4^2), (10^1, 5^2), (12^2, 11^0)\}$

For this execution, some coverd elements are the edges-s $(6^0, 5^1)$ and $(12^2, 11^0)$, the comm-c-use $(1^0, 7^1, avail)$, $(7^2, 8^1, avail)$, $(7^1, 8^2, queue)$.

To illustrate how the testing criteria can contribute to reveal faults, consider that the mutex semaphore was initialized with the value 0 or 2 on the main function (code line 17). This will cause a deadlock state or an inappropriate concurrent access to shared variables respectively. An execution that covers the required elements $(3^0, 6^1)$ and $(1^0, 7^2, avail)$, edges-s and comm-c-use respectively, will reveal the fault for the deadlock case. The execution of the required element comm-c-uses $(8^1, 8^2, queue)$ will reveal the fault for the case of inappropriate concurrent access. For both cases other required elements can also reveal these faults.

To illustrate a communication fault consider that *avail* was initialized with 1 (1^0) and all synchronizations are correct. This fault can be revealed with the execution of the required elements comm-c-use $(7^2, 7^2, avail)$ and edge-s $(9^2, 5^2)$. It will be necessary the execution of the PPCC sequence to reveal this fault, since the paths executed with the PCPC sequence do not reveal it.

5 Conclusion

Concurrent programs testing is not a trivial activity. This paper contributes in this context by addressing some of these problems for semaphore-based multithreading programs. The paper introduced both structural testing criteria to validate shared-memory parallel programs and a new model test to capture information about control, data, communication and synchronization from these programs. The paper also presents a post-mortem method based on timestamps to determine which communications (related with shared variables) happened in an execution. This information is important to establish the pairs of definition and use of the shared variables.

The proposed testing criteria are based on models of control and data flow and include the main features of the most used PThreads/ANSI C programs. The model considers communication, concurrency and synchronization faults among threads and also fault related to sequential aspects of each thread.

The use of the proposed criteria contributes to improve the quality of the test cases. The criteria offer a coverage measure that can be used in two testing procedures. Firstly, for generation of test cases, where these criteria can be used as guideline for test data selection. Secondly, for the evaluation of a test set. The

criteria can be used to determine when the testing activity can be ended and also to compare test sets.

The evolution of our work on this subject is directed to several lines of research: 1) development of a supporting tool for the introduced testing criteria (it is now being implemented); 2) development of experiments to refine and evaluate the testing criteria; 3) implementation of mechanisms to validate multithread programs that dynamically create threads; and 4) conduction of an experiment to evaluate the efficacy of the generated test data against *ad hoc* test sets.

References

[1] Yang, C.D., Pollock, L.L.: The challenges in automated testing of multithreaded programs. In: 14th Int. Conference on Testing Computer Software, pp. 157–166 (1997)
[2] Dongarra, J.J., Walker, D.W.: The quest for petascale computing. Computing in Science and Engineering 03(3), 32–39 (2001)
[3] Rapps, S., Weyuker, E.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering SE-11(4), 367–375 (1985)
[4] Taylor, R.N., Levine, D.L., Kelly, C.D.: Structural testing of concurrent programs. IEEE Trans. on Software Engineering 18(3), 206–215 (1992)
[5] Yang, C.S.D., Souter, A.L., Pollock, L.L.: All-du-path coverage for parallel programs. In: Young, M. (ed.) ISSTA 1998: Proc. of the ACM SIGSOFT Int. Symposium on Software Testing and Analysis, pp. 153–162 (1998)
[6] Lei, Y., Carver, R.: Reachability testing of concurrent programs. IEEE Trans. on Software Engineering 32(6), 382–403 (2006)
[7] Vergilio, S.R., Souza, S.R.S., Souza, P.S.L.: Coverage testing criteria for message-passing parallel programs. In: 6th LATW, Salvador, Ba, pp. 161–166 (2005)
[8] Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded Java programs. Concurrency and Computation: Practice and Experience 15(3–5), 485–499 (2003)
[9] Lei, Y., Carver, R.H., Kacker, R., Kung, D.: A combinatorial testing strategy for concurrent programs. Softw. Test., Verif. Reliab. 17(4), 207–225 (2007)
[10] Yang, C.S.D., Pollock, L.L.: All-uses testing of shared memory parallel programs. Softw. Test, Verif. Reliab. 13(1), 3–24 (2003)
[11] Lamport, L.: The implementation of reliable distributed multiprocess systems. Computer Networks 2, 95–114 (1978)
[12] Carver, R.H., Tai, K.C.: Replay and testing for concurrent programs. IEEE Softw. 8(2), 66–74 (1991)