Contribuições para o Teste de Software

Adenilso da Silva Simão

Contribuições para o Teste de Software

Adenilso da Silva Simão

Texto sistematizando o trabalho científico do candidato, 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.

São Carlos/SP
Janeiro/2011

Resumo

Este texto foi elaborado para a participação do autor no Concurso Público de Professor Livre Docente junto ao Departamento de Sistemas de Computação do Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (SSC/ICMC/USP). No texto, é apresentada uma sistematização das pesquisas e trabalhos realizados pelo autor, delineando seu foco de pesquisa em *Teste de Software*. Primeiramente, são descritas as contribuições na área de *Teste Baseado em Máquinas de Estados Finitos*, o qual concentra os principais esforços do autor e as publicações mais recentes. Em seguida, são apresentadas as contribuições na área de *Teste de Programas Paralelos* que caracteriza a segunda linha de atuação do autor. Por fim, são descritas as outras linhas de pesquisa nas quais o autor tem atuado.

Neste documento, são descritos os fundamentos e os pressupostos com as quais as abordagens têm sido exploradas pelo autor, destacando as contribuições e os desenvolvimentos realizados nos últimos seis anos de trabalho acadêmico, correspondendo ao período posterior à conclusão do doutorado. Nesse período, o autor publicou oito artigos completos em revistas, incluindo publicações na *IEEE Transactions on Computers*, *Oxford Computer Journal*, *IET Software* e *Computer Languages, Systems and Structures*, com Qualis A1, B1, B2 e B2, respectivamente. Publicou também 24 artigos em congressos da área, sendo 13 em eventos internacionais e 11 em eventos nacionais.

Abstract

This document was elaborated to fulfill the requirements of the author's application for a position of Associate Professor in Software Engineering, at the Computer Systems Department of the Institute of Mathematical Sciences and Computing, University of São Paulo (SSC/ICMC/USP). The text systematizes the author's research contribution, focused on studies about *Software Testing*. First, it presents the contributions on software testing based on Finite State Machines, which represents the core of the author's contributions and publications in recent years. Then, it presents the contributions on Parallel Program Testing, which represents his second main topic of investigation. Finally, it describes other research topics which the author has investigated.

This text describes the background and the assumptions which are the basis for the research done by the author, highlighting the contributions and developments accomplished in the last six years, i.e. after the conclusion of the Doctoral Thesis. In this period, the author published eight papers in journals, including *IEEE Transactions on Computers*, *Oxford Computer Journal*, *IET Software* e *Computer Languages, Systems and Structures*, evaluated as Qualis A1, B1, B2 e B2, respectively. He also published 24 papers in conferences, whereof 13 in international events and 11 in national ones.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Neste capítulo, é apresentado o contexto das linhas de pesquisa em que se inserem os trabalhos desenvolvidos pelo autor, bem como as motivações das atividades e pesquisas realizadas (Seção 1.1). Na Seção 1.2 é apresentada a organização do presente texto.

1.1 Contexto

O uso de software nas mais diversas áreas de aplicação impõe a necessidade de técnicas e ferramentas que auxiliem em seu desenvolvimento. O objetivo geral da Engenharia de Software é prover tais técnicas e ferramentas, buscando desenvolver produtos de alta qualidade e baixo custo.

Embora a Engenharia de Software proporcione métodos, técnicas e ferramentas para auxiliar na garantia da qualidade do produto de software desenvolvido, defeitos podem ser inseridos, o que traz a necessidade de uma etapa no desenvolvimento de software que tenha como objetivo minimizar a ocorrência de erros e riscos associados (Maldonado et al., 2004). Uma das atividades dessa etapa é a Verificação e Validação (V&V).

O Teste de Software é uma das atividades de V&V, a qual consiste na análise dinâmica do software com o objetivo de revelar a presença de defeitos no produto e, indiretamente, aumentar a confiança na qualidade desse produto. Um

teste bem sucedido é aquele que revela a presença de um ou mais defeitos até então não encontrados (Myers et al., 2004). Quando executado de forma sistemática e criteriosa, o teste contribui para aumentar a confiança de que o software apresenta os requisitos anteriormente estabelecidos, uma vez que, em geral, não é possível provar que um programa está isento de defeitos (Harrold, 2000; Weyuker, 1996).

De acordo com Pressman (2005), uma estratégia de teste de software integra módulos de projeto de casos de teste em uma série planejada de etapas, fornecendo um roteiro que descreve os passos a serem conduzidos. Essa estratégia deve ser flexível e, ao mesmo tempo, controlada, de forma a promover um planejamento razoável e acompanhamento gerencial à medida que o projeto avança. Uma estratégia de teste deve incorporar atividades, tais como: planejamento de teste, que é responsável por formular a maneira em que a atividade de teste será conduzida, como por exemplo, a escolha das técnicas e critérios a serem utilizados; projeto de casos de teste, o qual consiste na elaboração dos casos de teste a partir dos critérios estabelecidos; execução do teste, que conduz a aplicação dos casos de teste criados anteriormente; coleta e avaliação dos resultados do teste, a qual se tem um levantamento de como a atividade foi conduzida e os resultados obtidos (Pressman, 2005; Maldonado, 1991; Beizer, 1990).

Um dos pontos mais importantes e cruciais da atividade de teste é o projeto de casos de teste. Um caso de teste é um par ordenado composto pela entrada e pela saída esperada. Um conjunto de casos de teste forma um conjunto de teste. Segundo Myers et al. (2004), um bom caso de teste é aquele que tem alta probabilidade de encontrar um defeito ainda não descoberto. Porém, a construção do conjunto de teste não é trivial, uma vez que, na maioria dos casos, deve-se selecionar um conjunto específico e finito, já que se torna impraticável testar todo o domínio de entrada de um software. Para isso, tem-se o conceito de critério de teste, que tem como objetivo a seleção e/ou avaliação dos casos de teste, de forma a aumentar as possibilidades de revelar a presença de defeitos e estabelecer um nível elevado de confiança na correção do produto (Fabbri and Maldonado, 2001). Um critério de teste define requisitos de teste que um conjunto de teste deve satisfazer.

Técnicas de teste foram estabelecidas com o objetivo de encontrar o máximo de defeitos possíveis de um software. Essas técnicas são classificadas de acordo com a origem da informação que é utilizada para estabelecer os requisitos de testes (Maldonado, 1991). As principais técnicas de teste são:

Funcional: conhecida também como caixa-preta, considera o sistema como uma caixa fechada da qual não se tem conhecimento sobre sua implementação ou seu comportamento interno. No teste funcional, os testes são gerados somente considerando os valores de entrada e saída do sistema utilizando como base a sua especificação.

Estrutural: conhecida também como caixa-branca, estabelece os requisitos de teste baseados na estrutura interna do produto em teste. A geração dos testes considera as estruturas lógicas e funcionais implementadas, verificando se as funcionalidades e os resultados gerados estão de acordo com a especificação. Por ser baseado no conhecimento da estrutura interna da implementação, o testador deve ter acesso ao código fonte do programa, que é utilizado para gerar os casos de teste.

Baseada em Defeitos: estabelece os requisitos de teste explorando os defeitos típicos cometidos durante o desenvolvimento de software (DeMillo, 1980). Várias características do desenvolvimento de software devem ser consideradas quando se trata do teste baseado em defeitos, como a linguagem utilizada, ferramentas, tipo de software, entre outros.

Essas técnicas são em geral complementares, devendo ser aplicadas de forma estratégica em um programa para obter melhores resultados (Maldonado, 1991).

Para auxiliar no projeto de casos de teste, é importante que se tenha uma definição clara de o que é a saída esperada para uma dada entrada. Assume-se a existência de um *oráculo* capaz de determinar se o programa passou ou não no teste. Contudo, se o oráculo for um procedimento manual, a quantidade de testes que podem ser executados é limitada. Uma abordagem que tem sido empregada para, por um lado, ajudar na geração de casos de teste, e, por outro lado, simplificar a definição das saídas esperadas é a utilização de um modelo formal, com

semântica bem definida, que permite automatizar a tarefa de decidir se o teste produziu ou não a saída esperada. Coletivamente denominado *Teste Baseado em Modelos* (Pretschner and Philipps, 2004), essa abordagem permite que um volume maior de testes seja aplicado, pois, entre outras vantagens, automatiza a tarefa do oráculo. Dentre as diversas técnicas de teste baseado em modelos, as baseadas em máquinas de estados finitos vem recebendo grande atenção da comunidade acadêmica e da indústria há várias décadas.

O teste baseado em máquinas de estados finitos tem uma longa história, sendo que os primeiros trabalhos datam da década de 50 (Moore, 1956; Hennie, 1964). Trata-se, contudo, de uma área que continua sendo ativamente investigada (Hierons et al., 2009). Diversos métodos de geração têm sido propostos. Novos métodos geralmente incorporam avanços no entendimento das características que fazem com que os conjuntos de teste gerados apresentem propriedades desejadas.

A eficácia de uma estratégia de teste está diretamente relacionada com as características dos programas a serem testados. Por exemplo, programas que resolvem problemas numéricos devem ser testados de forma diferente de programas baseados em transações. Assim, é importante que estratégias de teste sejam investigadas em diversos paradigmas e técnicas de programação.

Programas paralelos são aqueles em que dois ou mais processos são executados simultaneamente. Programas paralelos adicionam um nível maior de complexidade durante a fases de projeto e implementação, pois devem ser levados em consideração detalhes de comunicação e sincronização entre processos, não determinismo, etc. Da mesma forma, o teste de programas paralelos deve levar em consideração características que podem impedir que técnicas de teste de programas tradicionais sejam aplicadas adequadamente. O teste de programas paralelos devem lidar com características que, em geral, não estão presentes no teste de programas tradicionais, tais como problemas de sincronização entre processos, *deadlocks* e *livelocks*, e o não determinismo na execução.

Os trabalhos desenvolvidos pelo autor após a conclusão do doutorado estão focados principalmente nas subáreas do teste baseado em modelos, em particular,

o teste baseado em máquinas de estados finitos, e no teste de programas paralelos.

De acordo com Maldonado (1991), as contribuições na área de Teste de Software podem ser divididas em:

Estudos Teóricos: Avançam o estado da arte melhorando o entendimento dos problemas da área, propondo novas abordagens e descobrindo as limitações teóricas inerentes às abordagens existentes. Por exemplo, pode-se determinar qual é a complexidade de um critério de teste, tanto em termos do custo para aplicá-lo quanto em termos do tamanho dos conjuntos de caso de teste necessários para satisfazê-lo.

Estudos Experimentais: Permitem que as diversas abordagens de teste sejam comparadas empiricamente, ou seja, por meio de experimentos com testadores ou artefatos gerados de forma a identificar o custo e a eficácia de cada uma.

Automatização: Consiste no desenvolvimento de ferramentas e ambientes que automatizem a atividade de teste. É de suma importância, pois aumenta a produtividade e a qualidade dos testes realizados.

Os trabalhos do autor, sistematizados neste documento, enquadram-se nessas três categorias.

1.2 Organização do Texto Sistematizado

Neste texto sistematizado é apresentada uma descrição das principais contribuições resultantes das atividades de pesquisa realizadas pelo autor. No Capítulo 2 são descritos os trabalhos desenvolvidos nas linhas de pesquisa, dando ênfase ao relacionamento entre as pesquisas realizadas. Na Seção 2.1 são apresentados os trabalhos relacionados ao teste baseado em máquinas de estados finitos, ao passo que na Seção 2.2 são apresentados os resultados relacionados ao teste de programas paralelos. Na Seção 2.3 são apresentados os trabalhos desenvolvidos na área de teste de software, mas que não se enquadram nas duas linhas anteriores, tais

como o teste baseado em defeitos e o teste funcional. No Capítulo 3 são discutidas as conclusões e indicados os trabalhos futuros e em andamento. Nos Apêndice A e B são apresentados os principais conceitos e as definições relacionadas ao teste baseado em máquinas de estados finitos e ao teste de programas paralelos, respectivamente. Por fim, nos Apêndices C a J são incluídas as publicações mais relevantes, resultantes do trabalho aqui reportado.

Capítulo 2

Contribuições ao Teste de Software

A seguir, são resumidos os trabalhos desenvolvidos pelo autor ou sob sua orientação após a conclusão do doutorado. Os trabalhos serão classificados de acordo com o enfoque principal. As publicações relacionadas serão indicadas ao longo do texto e sumarizadas ao final. Os trabalhos foram divididos em três grupos, correspondentes às subseções desse capítulo. Primeiramente, são apresentados os trabalhos relacionados ao teste baseado em máquinas de estados finitos, que representa a maior parte dos trabalhos desenvolvidos pelo autor. Em seguida, apresenta-se o teste de programas paralelos, que representa uma área de investigação que o autor tem desenvolvido, podendo ser considerada como sua segunda principal área de atuação. Por fim, são apresentados trabalhos que são relevantes desenvolvidos em outras linhas, todas relacionadas a diferentes aspectos do teste de software.

Os trabalhos são apresentados de forma resumida. Apenas os conceitos principais são apresentados, assim como os aspectos que distinguem os trabalhos e caracterizam sua contribuição principal. Informações adicionais sobre os trabalhos são incluídas nos apêndices; definições formais, exemplos e detalhes podem ser lá encontrados. A referência de cada publicação é incluída em uma nota de rodapé, usando a seguinte convenção. Os círculos correspondem às publicações

em revista, enquanto que os quadrados correspondem às publicações em conferências. As publicações nacionais são apresentadas com linha tracejada. As oito publicações mais relevantes são apresentadas com fundo cinza; cópias dessas publicações podem ser encontradas nos Apêndices C a J.

2.1 Teste Baseado em Máquinas de Estados Finitos

Nesta seção, são apresentados os resultados obtidos na área de pesquisa do teste baseado em máquinas de estados finitos. Trata-se da área em que o autor tem atuado mais diretamente após a conclusão de seu doutorado. Deve-se destacar que no período de Ago/2008 a Jul/2010, o autor realizou um estágio de pós-doutoramento junto ao *Centre de Recherche Informatique de Montreal* (CRIM), em colaboração com o pesquisador Alexandre Petrenko, o que contribuiu para consolidar os resultados que vinham sendo desenvolvidos.

As pesquisas desenvolvidas nessa linha estão no contexto do teste baseado em modelos, que busca uma forma automatizada de gerar casos de teste a partir de uma especificação ou modelo. Embora alguns autores afirmem que o teste é sempre baseado em modelos, dado que modelos mentais implícitos são usados para guiar os testes (Binder, 1999), a ideia do teste baseado em modelos é utilizar modelos explícitos (Pretschner and Philipps, 2004). Utting and Legeard (2006) definem o teste baseado em modelos como automação do projeto de testes caixa-preta em que, dado um modelo de teste adequado, sequências de teste podem ser geradas e transformadas em *scripts* executáveis.

O modelo de teste pode ser construído manualmente, derivado de alguma especificação de requisitos ou fonte de conhecimento sobre o sistema, codificando o comportamento esperado de uma implementação chamada de sistema sob teste (*System Under Test* - SUT). É importante que a técnica de modelagem selecionada para o teste baseado em modelos seja formal (em outras palavras, bem definida sintática e semanticamente), pois a presença de modelos ou especificações formais pode levar a um teste mais eficiente e efetivo (Hierons et al., 2009). Segundo Utting and Legeard (2006), um modelo é formal se possui um significado preciso e não ambíguo, representando o comportamento de uma forma compreensível e

manipulável por ferramentas. Pela necessidade de validar o modelo, esse deve ser mais simples que o SUT, ou, no mínimo, mais fácil de verificar, modificar e manter (Utting et al., 2006). Entretanto, o modelo deve ser suficientemente preciso para servir como base para a geração de casos de teste significativos.

No teste baseado em modelos, a representação por meio de Máquinas de Estados Finitos (MEFs) (Gill, 1962) vem sendo frequentemente utilizada devido à sua simplicidade e capacidade de modelar sistemas, principalmente na modelagem de protocolos de comunicação e sistemas reativos. Além disso, o teste baseado em MEFs pode ser aplicado em outros tipos de sistemas, como sistemas orientados a objetos (Hong et al., 1995) e sistemas Web (Andrews et al., 2005). Outra vantagem do uso de MEFs segue do fato de existirem vários métodos de geração de sequências de teste, oferecendo apoio e direcionamento nos testes gerados e executados.

As MEFs são uma técnica formal que se tem mostrado bastante útil para tratar o comportamento de sistemas e para ser utilizada no teste de software. Essa técnica é muito utilizada para modelar o comportamento de sistemas reativos, pois esses são essencialmente dirigidos a eventos e dominados por controle. Além disso, as MEFs possuem uma gama de aplicações bastante grande e genérica, podendo ser utilizadas na modelagem de vários tipos de sistemas. Sendo assim, seus modelos são aplicáveis em diversos contextos, como por exemplo, em protocolos de comunicação, sistemas reativos, circuitos elétricos, entre outros.

Segundo Gill (1962), uma MEF é uma máquina hipotética composta por estados e transições. Cada transição liga um estado a a um estado b (a e b podem ser o mesmo estado). A cada instante, uma máquina pode estar em apenas um de seus estados, o que caracteriza uma máquina determinística, caso contrário é uma máquina não determinística. Em resposta a um evento de entrada, a máquina gera um evento de saída e executa uma transição. Tanto o evento de saída gerado quanto o novo estado são definidos unicamente em função do estado atual e do evento de entrada (Davis, 1988).

A utilização de MEFs no contexto do teste de software vem sendo investigado há várias décadas, sendo que os primeiros trabalhos datam das décadas de 50 (Moore, 1956) e 60 (Hennie, 1964). Dentre os métodos mais conhecidos, pode-

se destacar os métodos DS (Gonenc, 1970), W (Chow, 1978), UIO (Sabnani and Dahbura, 1988), Wp (Fujiwara et al., 1991), HSI (Petrenko et al., 1993; Luo et al., 1994), H (Dorofeeva et al., 2005a) e *State Counting* (Petrenko and Yevtushenko, 2005).

Em geral, a aplicação dos métodos de geração requer que as MEFs possuam certas propriedades, sendo que diferentes métodos podem requerer diferentes conjuntos de propriedades. Dessa forma, os diversos métodos de geração de sequências de teste a partir de MEFs podem ser classificados com base em três características:

- Aplicabilidade, que se refere às propriedades necessárias para aplicação do método.
- Completude, que se refere à classe de defeitos que o método garante revelar.
- Tamanho dos conjuntos e número de sequências de teste geradas.

O custo de aplicação dos métodos pode ser calculado em relação ao custo de geração das sequências de teste e ao custo da execução. Por exemplo, um método pode ser eficiente para gerar as sequências de teste, porém se as sequências geradas forem muito grandes, seu custo de execução é alto, o que pode torná-lo ineficiente. O custo de execução das sequências de teste é normalmente o fator dominante quando se avalia o custo da aplicação de um método. Sendo assim, o tamanho do conjunto de sequências de teste é geralmente utilizado para comparar o custo de aplicação do método. Além disso, o número de sequências geradas pelos métodos também é um fator de influência no custo do teste. Em geral, assume-se a existência de uma operação *reset*, que leva tanto a MEF quanto sua implementação ao seu estado inicial. A operação *reset* deve ser inserida no início de cada sequências do conjunto de teste; portanto, o número de operações *resets* é igual ao número de sequências de um conjunto de teste.

Na geração de testes a partir de MEFs, assume-se que a implementação pode ser modelada como uma MEF contida em um domínio de defeitos. Essa hipótese, conhecida como hipótese de teste, é necessária para que um conjunto finito de testes possa ser gerado (Chow, 1978; Ural et al., 1997; Hierons and Ural, 2006;

Hennie, 1964). O teste baseado em MEFs consiste na geração de um conjunto de sequências de teste cujo objetivo é encontrar o máximo de defeitos em uma implementação. Dessa forma, é possível verificar se a implementação da MEF está de acordo com sua especificação. Dada uma MEF M com n estados, $\Im_m(M)$ denota o domínio de defeitos definido pelo conjunto de todas as MEFs com o mesmo alfabeto de entrada e no máximo m estados, utilizado por grande parte dos métodos de geração, como por exemplo, os métodos W (Chow, 1978), Wp (Fujiwara et al., 1991), HSI (Petrenko et al., 1993; Luo et al., 1994), H (Dorofeeva et al., 2005a), entre outros. De acordo com Chow (1978), os defeitos são classificados em:

Defeito de transferência: transição atinge estado incorreto.

Defeito de saída: transição gera uma saída incorreta.

Estados faltantes: os estados da implementação devem ser aumentados para torná-la equivalente à especificação.

Estados extras: os estados da implementação devem ser reduzidos para torná-la equivalente à especificação.

Todos esses defeitos podem ser modelados por MEFs pertencentes a $\Im_m(M)$, caso o parâmetro m seja escolhido adequadamente. Para que o teste em MEFs possa ser realizado, deve-se estimar o número m de estados da implementação, sendo que quanto melhor for essa estimativa, melhor será o conjunto de teste obtido. Os métodos de geração de casos de teste consideram que a MEF possui no máximo m estados, tal que m seja maior ou igual a n (número de estados da especificação). A partir dessa informação, a implementação estará de acordo com sua especificação se não possuir defeitos de transferências nem defeitos de saída. Para fins de entendimento deste trabalho, será considerado o teste de MEFs em que o número de estados é igual o da implementação, ou seja, $n = m$ e o defeito de estados extras não é considerado. Dessa forma, $\Im(M)$ contém todas as MEFs que modelam os defeitos que se encaixam no contexto deste trabalho.

Um conjunto de sequências de teste T é *n-completo*, ou simplesmente *completo*, se para cada MEF $N \in \Im(M)$ tal que N e M são distinguíveis, existe uma sequência pertencente a T que distingue N de M. Ou seja, se o conjunto é completo, ele

é capaz de revelar todos os defeitos de uma implementação de M que possa ser modelada por uma MEF de $\Im(M)$.

Projeto Plavis

A pesquisa com a geração de testes baseados em MEFs era um dos objetivos do projeto Plavis (*Platform for Software Validation & Integration on Space Systems*, CNPq Processo n^o 473396/2003-3 Vigência: 01/06/2003 a 30/09/2005, prorrogado até 30/09/2006), no qual o autor participou após a conclusão do doutorado. O projeto contava com a participação de pesquisadores de diversas universidades brasileiras e francesas, além do Instituto Nacional de Pesquisas Espaciais (INPE). No contexto desse projeto, foi desenvolvido um ambiente para integrar diversas ferramentas relacionadas ao teste e a MEFs desenvolvidas pelos membros do projeto. Esse ambiente, denominado Plavis/FSM (Simão et al., 2005)[1], foi utilizado em alguns cursos de graduação no ICMC e continua sendo utilizado em cursos de pós-graduação no INPE. O ambiente Plavis/FSM serviu como base para trabalhos de conclusão de curso e de iniciação científica. Contudo, no contexto dos trabalhos desenvolvidos pelo autor, o projeto foi muito relevante, pois foi por meio desse projeto que o autor iniciou a investigação dos problemas clássicos referentes a essa linha de pesquisa.

A arquitetura da Plavis/FSM é apresentada na Figura 2.1; a Plavis/FSM foi desenvolvida como uma aplicação Web e está disponível para uso remoto. A principal motivação foi permitir que a ferramenta pudesse ser utilizada sem a necessidade de instalação. A incorporação de as ferramentas integradas é feita por meio de adaptadores; foram integradas as ferramentas Proteum/FSM (Fabbri et al., 1999), Condado (Martins et al., 1999) e MGASet (Candolo et al., 2001).

Os primeiros trabalhos orientados pelo autor nessa linha foram dois trabalhos de iniciação científica, em 2005. Primeiramente, no trabalho de Leonardo Filonones Teixeira, foi desenvolvido um mecanismo de filtro de casos de teste. Em geral, o número de casos de teste gerados pelos métodos integrados na Plavis/FSM é

[1] **A. S. Simão**, A. M. Ambrosio, S. C. P. F. Fabbri, A. S. Amaral, E. Martins, J. C. Maldonado. Plavis/FSM: an Environment to Integrate FSM-based Testing Tools. In: Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software, 2005. p. 1-6, Uberlândia, MG

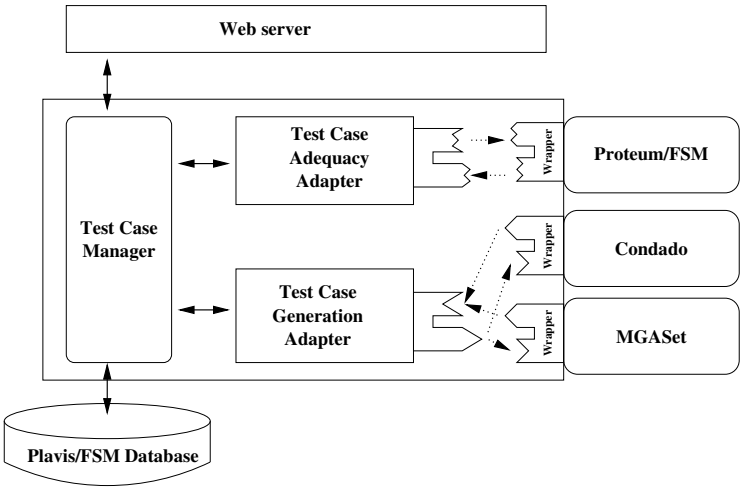


Figura 2.1: Arquitetura da Plavis/FSM.

muito alto. Dessa forma, nem sempre é possível que todos sejam aplicados. Um subconjunto dos casos de teste deve ser selecionado. Vários critérios podem ser utilizados para a seleção, tais como casos que passam por determinados estados, usam uma determinada entrada ou saída, ou exercitam uma transição. Assim, o testador pode definir critérios, por meio de alguns parâmetros simples, para a seleção de casos de teste gerados a partir de MEFs. O filtro foi então incorporado à Plavis/FSM. Ainda que seja um trabalho relativamente simples, os estudos realizados durante a iniciação científica foram base para definir metas de pesquisa a longo prazo, tais como os trabalho de minimização de conjuntos de teste desenvolvidos posteriormente em trabalhos de mestrado.

No trabalho de iniciação científica desenvolvido por Jorge Francisco Cutigi, foi implementado o método HSI (Petrenko et al., 1993; Luo et al., 1994). O método era relevante no contexto do projeto Plavis devido ao fato de que os métodos até então integrados exigiam que a MEF fosse completamente especificada (ou seja, em cada estado, existe uma transição para cada entrada), ao passo que muitas das MEFs utilizadas no projeto eram parciais (ou seja, não completas). O método HSI pode ser aplicado a MEFs parciais e nesse trabalho de iniciação científica ele foi estudado e implementado. Novamente, esse trabalho foi importante para estabelecer as bases para diversos trabalhos futuros. Principalmente, foi investigada a

noção de completude de casos de teste, a qual passou a figurar em praticamente todos os trabalhos posteriores nessa linha.

Minimização de Conjuntos Completos

Em 2006, o autor juntou-se ao corpo de orientadores do Programa de Pós-Graduação de Ciências de Computação e Matemática Computacional do ICMC/USP. O primeiro trabalho de mestrado nessa linha de investigação foi o trabalho desenvolvido por Lúcio Felipe de Mello Neto. O tema investigado foi a minimização de casos de teste, de forma a manter a completude na capacidade de detecção de defeitos. Primeiramente, foi identificado um trabalho (Dorofeeva et al., 2005a) que definia um conjunto de condições de suficiência para completude de casos de teste. Em geral, os métodos de geração garantem por construção que o conjunto de testes obtidos são completos. Contudo, poucos trabalhos investigavam como um conjunto arbitrário de sequências pode ser analisado para verificar se ele é ou não completo, sendo que os trabalhos de Petrenko et al. (1996) e Yao et al. (1994) eram os únicos encontrados na literatura até então. O trabalho de Dorofeeva et al. (2005a) apresenta um avanço, no sentido de identificar condições de suficiência mais flexíveis. Apesar de os autores desse trabalho apenas as utilizarem para propor um novo método (o método H) que gera conjuntos completos por construção, durante o mestrado de Mello Neto foi observado que tais condições poderiam ser utilizados para a minimização de conjuntos de forma a manter a completude.

Foi desenvolvido um algoritmo, baseado nas condições de suficiência definidas por Dorofeeva et al. (2005a), que, dados uma MEF e um conjunto de teste que satisfaz tais condições (e, portanto, é completo), seleciona um subconjunto que ainda satisfaça tais condições. Resultados preliminares do algoritmo foi publicado em (Mello Neto and Simão, 2007)[2]. A extensão do algoritmo e os estudos experimentais realizados para avaliá-lo foram posteriormente publicados em

[2] L. F. Mello Neto, **A. S. Simão**. Minimização de Conjuntos de Casos de Teste por Meio de Condições de Suficiência. In: The 1st Brazilian Workshop on Systematic and Automated Software Testing. p. 55-62. João Pessoa, PB, 2007.

(Mello Neto and Simão, 2008)[3]. Um possível cenário de utilização do algoritmo é quando já existe um conjunto de teste (obtido de forma *ad hoc*), mas deseja-se também a garantia de detecção de defeitos oferecida pelos métodos completos. Assim, o conjunto inicial pode ser complementado com um conjunto completo gerado pelos Métodos W, Wp, HSI ou H, e o algoritmo desenvolvido por de Mello Neto se encarregaria de remover sequências desnecessárias.

O trabalho de Mello Neto motivou a investigação mais aprofundada do trabalho de Dorofeeva et al. (2005a). Foi desenvolvida a iniciação científica por José Augusto Stuchi, cujo objetivo era a implementação do método H, proposto nesse artigo. Apesar de ser apenas um trabalho de iniciação científica, as investigações realizadas durante o desenvolvimento desse trabalho resultou em um melhor entendimento das limitações das condições de suficiência propostas e serviu de base para trabalhos futuros. Foi realizado um experimento com o método H e mostraram que os conjuntos gerados por esse método é, em média, 66% do tamanho dos conjuntos gerados pelo método HSI (Petrenko et al., 1993; Luo et al., 1994).

Um dos passos do método H que possui impacto direto diz respeito à escolha de sequências de separação de dois estados (Dorofeeva et al., 2005a). Dados dois estados, uma sequência de separação é uma sequência de entrada tal que esses estados produzam resultados diferentes (ou seja, saídas diferentes). O trabalho de conclusão de curso desenvolvido por Guilherme Botelho Diniz Junqueira tinha como objetivo identificar estratégias que permitissem selecionar as sequências de distinção que levariam ao menor acréscimo no tamanho atual do conjunto de teste. Assim, pôde-se estudar como diferentes formas de selecionar as sequências de distinção podem ser utilizadas e como essas formas influenciam o tamanho final do conjunto de teste. Os resultados desses estudos foram incorporados em trabalhos futuros.

[3] L. F. Mello Neto, **A. S. Simão**. Test Suite Minimization Based on FSM Completeness Sufficient Conditions. In: The 9th IEEE Latin-American Test Workshop. p. 93-98. Puebla, Mexico, 2008. **(Qualis B3)**

Condições de Suficiência

Os trabalhos de Mello Neto (em nível de mestrado), Stuchi (em nível de iniciação científica) e de Junqueira (em nível de conclusão de curso) motivaram o estudo de condições de suficiência para completude de conjuntos de casos de teste gerados a partir de MEFs. Observou-se que melhorias nessas condições podem ser utilizadas em diversos contexto. Por exemplo, as condições propostas por Dorofeeva et al. (2005a) foram a base para a proposição de um novo método (especificamente, o Método H, que foi estudado por Stuchi) e para implementar o algoritmo de minimização no trabalho de mestrado de Mello Neto. Assim, melhorias adicionais nas condições poderiam resultar em métodos mais eficazes.

Um conjunto separado de condições de suficiente é apresentado em (Ural et al., 1997). Essas condições aplicam-se para sequências de verificação, que correspondem a conjuntos completos formados por uma única sequência. Sequências de verificação são relevantes, pois não utilizam a operação de *reset*, a qual em algumas situações pode ser custosa de ser utilizada.

As condições de suficiência propostas em (Dorofeeva et al., 2005a) e (Ural et al., 1997) são ortogonais: umas não podem ser derivadas das outras. Por outro lado, todas as demais condições (por exemplo, (Petrenko et al., 1996) e (Aho et al., 1991)) propostas na literatura podem ser derivadas de um ou outro conjunto. Em (Simão and Petrenko, 2010a)[4], foi definido um conjunto de condições de suficiência que generalizam ambos os conjuntos. Por consequências, as condições propostas generalizam todas as condições propostas na literatura até o momento. Uma importante contribuição das condições foi demonstrar que tantos os métodos baseados em conjuntos de caracterização (tais como, W, Wp, HSI e H) e os baseados em sequências de distinção podem ser conciliados. Na Figura 2.2 ilustra-se a relação entre as condições de (Dorofeeva et al., 2005a), (Ural et al., 1997) e (Simão and Petrenko, 2010a). Existem conjuntos que satisfazem as condições de (Dorofeeva et al., 2005a), mas não as de (Ural et al., 1997); similarmente, existem conjuntos que satisfazem as condições de (Ural et al., 1997), mas não as

[4] **A. S. Simão**, A. Petrenko. Checking Completeness of Tests for Finite State Machines. IEEE Transactions on Computers, v. 59, p. 1023-1032, 2010. **(Qualis A1)**

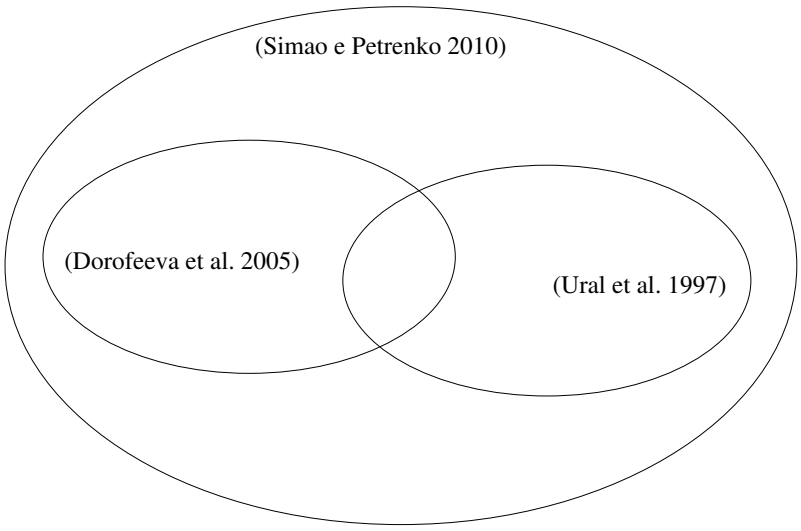


Figura 2.2: Condições de Suficiência.

condições de (Dorofeeva et al., 2005a). Por outro lado, foi demonstrado que todos os conjuntos que satisfazem quer as condições de (Dorofeeva et al., 2005a) quer as condições de (Ural et al., 1997) também satisfazem as condições de (Simão and Petrenko, 2010a). Além disso, demonstrou-se que existem conjuntos que satisfazem as condições de (Simão and Petrenko, 2010a) e não satisfazem as condições de (Dorofeeva et al., 2005a) e nem as de (Ural et al., 1997).

As condições propostas em (Simão and Petrenko, 2010a) foram a base para dois trabalhos de mestrado. O artigo estava em avaliação enquanto os trabalhos estavam sendo desenvolvidos. No trabalho de Jorge Francisco Cutigi, foi elaborado um método de minimização de conjuntos de teste completos. O método pode ser aplicado com dois objetivos: redução no número de sequências do conjunto ou redução no tamanho total do conjunto. Diferentemente do trabalho de Mello Neto, as condições utilizadas por Cutigi permitiram que a redução no número de sequências fosse muito expressiva (Cutigi et al., 2010)[5], em média 80%.

[5] J. F. Cutigi, P. H. Ribeiro, **A. S. Simão**, S. R. S. Souza. Redução do Número de Sequências no Teste de Conformidade de Protocolos. In: XI Workshop de Testes e Tolerância a Falhas, p. 105-117, 2010, Gramado, RS. **(Qualis B5)**

Todos os métodos de geração, exceto o método *State Counting* (Petrenko and Yevtushenko, 2005), só podem ser aplicados a MEFs reduzidas (ou seja, sem estados equivalentes). Para ser aplicado a uma MEF não reduzida, essa deve portanto ser inicialmente convertida para uma forma reduzida. Enquanto a redução de MEFs completas pode ser realizada por algoritmos polinomiais, a redução de MEFs parciais é um problema de alto custo computacional. No trabalho de Alex Donizeti Betez Alberto, foi proposto um algoritmo para redução de MEFs parciais. O algoritmo (Alberto and Simão, 2009)[6] foi experimentalmente comparado com algoritmos encontrados na literatura. Observou-se o que, por um lado, os resultados eram comparáveis ao método que apresentavam os melhores resultados enquanto que, por outro lado, o tempo gasto para a redução foi consideravelmente menor.

Sequências de Verificação

A geração de sequências de verificação é um tópico que vem sendo investigado desde os trabalhos seminais de Hennie (Hennie, 1964). O objetivo é produzir uma sequência de entrada que forme um conjunto completo unitário. Em (Hennie, 1964) foi demonstrado que se a MEF possui uma sequência de distinção, é possível gerar uma sequência de verificação. Contudo, não foi apresentado nenhum método sistemático. Em (Gonenc, 1970), um algoritmo baseado em grafos é proposto, o qual procura sistematizar o método proposto por Hennie. O interesse na geração de sequências foi retomado a partir dos trabalhos de (Ural et al., 1997). Em essência, esse trabalho é baseado no trabalho de (Gonenc, 1970), modelando-o como um problema de otimização em grafo, a saber o problema do Carteiro Rural Chinês (Aho et al., 1991). Diversos trabalhos seguiram a mesma linha, tais como (Hierons and Ural, 2002; Chen et al., 2005; Ural and Zhang, 2006; Hierons and Ural, 2006).

Apesar de diversos trabalhos terem sido desenvolvidos com base no trabalho de (Ural et al., 1997), os ganhos na redução no tamanho das sequências de ve-

[6] A. D. B. Alberto, **A. S. Simão**. Minimization of Incompletely Specified Finite State Machines Based on Distinction Graphs. In: The 10th Latin-American Test Workshop. p. 1-6, , Buzios, RJ, 2009. **(Qualis B3)**

rificação são pequenos. Parte do problema pode ser atribuído ao fato de que os modelos de otimização se concentram em minimizar sequências de transferência, que em geral correspondem a uma pequena parte do tamanho total do problema. Dados dois estados, uma sequência de transferência é uma sequência de entrada que leva a MEF de um estado a outro. Uma abordagem diferente, baseada em busca local, foi proposta em (Simão and Petrenko, 2008)[7]. Diferentemente dos trabalhos baseados em (Ural et al., 1997), em vez de procurar modelar a geração como um problema de otimização, o método proposto busca a cada passo adicionar o mínimo de entradas necessário para verificar uma transição. O trabalho foi experimentalmente comparado com os dois melhores métodos baseados no trabalho de (Ural et al., 1997), a saber, (Chen et al., 2005; Hierons and Ural, 2006); em 75% dos casos, o método proposto gerou sequências menores do que as geradas pelo método proposto por (Chen et al., 2005); em todos os casos, o método proposto gerou sequências menores do que as geradas pelo método proposto por (Hierons and Ural, 2006).

Em (Simão and Petrenko, 2009)[8], foi demonstrado que em alguns casos pode-se utilizar as sequências de distinção em apenas algumas partes, enquanto que em outras utilizam-se sequências de identificação de estado, tais como nos métodos baseados no W. É importante observar que já em (Hennie, 1964) foi mencionado que seria possível evitar a utilização das sequências de distinção para geração de sequências de verificação. Contudo, ainda não havia um método sistemático que indicasse como isso pudesse ser feito. Assim, o trabalho desenvolvido corresponde a uma importante contribuição teórica. A contribuição prática, por outro lado, ainda não está clara, pois não foi possível identificar qual é o ganho na redução do tamanho das sequências de verificação geradas.

---

[7] **A. S. Simão**, A. Petrenko. Generating Checking Sequences for Partial Reduced Finite State Machines. In: The 20th IFIP Int. Conference on Testing of Communicating Systems (TESTCOM), p. 153-168, Tokyo, Japão, 2008. **(Qualis B3)**

[8] **A. S. Simão**, A. Petrenko. Checking Sequence Generation Using State Distinguishing Subsequences. In: The 5th Workshop on Advances in Model Based Testing. p. 1-10, Denver, USA, 2009.

No trabalho de mestrado de Paulo Henrique Ribeiro, um método baseado em algoritmos genéticos foi proposto para a geração de sequências de verificação (Ribeiro et al., 2009)[9]. Uma vez que condições de suficiência propostas generalizam também as condições propostas por (Ural et al., 1997), elas podem ser utilizadas para identificar quando uma sequência é uma sequência de verificação. Foi proposto então um método que, por meio de várias iterações e da seleção das sequências mais aptas, busca produzir a menor sequência possível. Apesar de realmente obter sequências de verificação menores, observou-se que o custo de aplicação é alto e os ganhos são relativamente pequenos. Dessa forma, outras estratégias de geração devem ser desenvolvidas. Ainda assim, pôde-se observar que as condições propostas em (Simão and Petrenko, 2010a) são realmente melhores do que as de (Ural et al., 1997), uma vez que quando o método foi alterado para utilizar estas condições no lugar daquelas, obteve-se sequências 12,7% maiores.

Melhorias em Métodos de Geração Existentes

Foram investigados também possíveis generalizações e melhorias dos métodos clássicos de geração. A investigação tinha como objetivo aumentar a aplicabilidade dos métodos (ou seja, permitir que fossem aplicados a uma classe maior de MEFs) ou reduzir o tamanho dos conjuntos gerados.

Em (Bonifácio et al., 2008a)[10], foi proposta uma generalização do método W, na qual não é requerido que a MEF possua um conjunto de caracterização. Contudo, é necessário que se tenha um conjunto de sequências e que se saiba em quantas classes esse conjunto particiona a implementação. Tais conjuntos podem ser obtidos por meio de teste de regressão ou quando padrões de projeto e implementação podem ter sido utilizados.

---

[9] P. H. Ribeiro, J. F. Cutigi, **A. S. Simão**. Geração de Sequências de Verificação baseada em Algoritmos Genéticos. In: The 3rd Brazilian Workshop on Systematic and Automated Software Testing. p. 61-70, Gramado, RS, 2009.

[10] A. L. Bonifácio, A. Moura, **A. S. Simão**. A Generalized Model-based Test Generation Method. In: The 6th IEEE International Conferences on Software Engineering and Formal Methods, p. 139-148, Cape Town, Africa do Sul, 2008. **(Qualis B2)**

Em geral, os conjuntos gerados são completos considerando todas as MEFs com no máximo o mesmo número de estados como domínio de defeitos. Contudo, domínios alternativos também podem ser considerados. Um domínio de defeitos que foi utilizado em diversos trabalhos permite que a MEF que modela a implementação possua estados extras, tais como as extensões dos métodos W, Wp, HSI e H. O limite inferior no tamanho dos conjuntos é exponencial em relação ao número de estados extras. Apesar de não ser possível reduzir esse limite, foi identificado que parte do tamanho dos conjuntos é na verdade devida ao excesso de prefixos comuns que são utilizados. Em (Simão et al., 2009c)[11], foi desenvolvido uma abordagem que permite que diversos prefixos sejam eliminados. Foi proposto então o método SPY, que é uma generalização do método HSI no caso de implementações com estados extras. Foi demonstrado experimentalmente que o método SPY gera conjuntos em geral 40% menores que o método HSI.

O trabalho publicado em (Simão and Petrenko, 2010b)[12] traz três contribuições principais. Primeiramente, investigou-se o domínio de defeitos correspondentes ao caso no qual a implementação pode ter no máximo um número menor de estados do que a especificação. Apesar de ser um domínio relativamente simples, trata-se do primeiro método que é capaz de gerar conjuntos completos para tal domínio. Em segundo lugar, a geração não precisa necessariamente começar do zero; pode-se iniciar a geração a partir de um conjunto já existente. Dessa forma, os conjuntos podem ser gerados incrementalmente. Observe que isso somente é possível devido à combinação com a contribuição anterior. Por fim, as condições propostas em (Simão and Petrenko, 2010a) foram generalizadas, de forma a poderem ser aplicadas a outros domínios. Apesar de apenas o domínio formado por implementações com no máximo um número menor de estados do que a especificação, as condições apresentadas nesse trabalho podem ser futura-

[11] **A. S. Simão**, A. Petrenko, N. Yevtushenko. Generating Reduced Tests for FSMs with Extra States. In: The 21st IFIP Int. Conference on Testing of Communicating Systems (TESTCOM). p. 129-147, Eindhoven, Holanda, 2009. **(Qualis B3)**

[12] **A. S. Simão**, A. Petrenko. Fault Coverage-Driven Incremental Test Generation. Computer Journal, v. 53, p. 1508-1522, 2010. **(Qualis B1)**

mente generalizadas para domínios específicos, tais como implementações das quais se sabe que algumas transições estão corretamente implementadas.

Testes Baseado em Verificadores de Modelos

Embora a geração de casos de teste a partir de MEFs ser um tópico bastante investigado, muitos sistemas não podem ser facilmente modelados se não forem incluídos mecanismos para a inclusão de recursos que permitam descrever como variáveis são manipuladas. Diversas extensões às MEFs para incluir tais recursos têm sido propostas, dando origem às MEFs Estendidas (MEFEs). Em (Bonifácio et al., 2006)[13], foi investigado como técnicas de verificação de modelos podem ser utilizadas para orientar a geração de casos de teste a partir de MEFEs adicionadas com informações de tempo. Em geral, muitos problemas relacionados à atividade de teste, tais como a distinção entre dois estados da MEFEs ou mesmo se um determinado estado é alcançável, são indecidíveis. Dessa forma, a aplicação de técnicas de teste utilizadas em MEFs não são facilmente aplicáveis a MEFEs. As técnicas de verificação de modelos são utilizadas para verificar se uma determinada propriedade, especificada por meio de uma lógica temporal, é válida para um determinado modelo. Caso não seja válida, um contraexemplo é produzido. Se a propriedade a ser verificada é cuidadosamente definida para refletir uma propriedade indesejada do sistema, o contraexemplo pode ser utilizado como base para a construção de casos de teste para testar a presença dessa propriedade no sistema. Uma versão estendida desse trabalho foi publicado em (Bonifácio et al., 2008b)[14].

[13] A. L. Bonifacio, **A. S. Simão**, A. Moura, J. C. Maldonado. Conformance Testing by Model Checking Timed Extended Finite State Machines. In: Simpósio Brasileiro de Métodos Formais. p. 43-58, Natal, RN, 2006. **(Qualis B3)**

[14] A. L. Bonifacio, A. Moura, **A. S. Simão**, J. C. Maldonado. Towards Deriving Test Sequences by Model Checking. Electronic Notes in Theoretical Computer Science, v. 195, p. 21-40, 2008. **(Qualis B2)**

Estudos Experimentais

Em alguns casos, é possível identificar que um método de geração produz conjuntos que são comprovadamente menores do que outros métodos. Contudo, em várias situações, os métodos são teoricamente incomparáveis: não é possível determinar qual método gera os menores conjuntos. Nesses casos, estudos experimentais são importantes (Dorofeeva et al., 2005b).

Em (Simão et al., 2007)[15], foi investigado qual é o comportamento típico de diversos critérios de cobertura para MEFs. Em geral, apenas os limites teóricos do tamanho dos conjuntos de teste gerados pelos diversos métodos são conhecidos. Na maior parte dos casos, tais limites são quadráticos ou cúbicos em função do número de estados da máquina. Contudo, mostrou-se no trabalho desenvolvido que para MEFs geradas aleatoriamente tais limites são em média muito menores. Por exemplo, enquanto o limite teórico para o método H é $O(n^3)$, onde n é o número de estados da MEF, os estudos experimentais apontam que em geral o tamanho do conjunto é $O(n^{1.4})$. Estes dados são importantes para que o testador possa ter subsídios para definir estratégias efetivas de teste. Uma extensão desse trabalho com a comparação do tamanho de conjuntos completos foi publicada em (Simão et al., 2009b)[16].

No trabalho de mestrado de Flávio Dusse (co-orientado pelo autor), foi investigado como a comparação do critérios de cobertura poderia ser melhorada com a inclusão da Análise de Mutantes. A conclusão principal desse trabalho foi de que o escore de mutação de um critério é diretamente relacionado ao tamanho médio dos conjuntos adequados (Dusse et al., 2009)[17]. Esse resultado confirma a intui-

[15] **A. S. Simão**, A. Petrenko, J. C. Maldonado. Experimental Evaluation of Coverage Criteria for FSM-based Testing. In: Simpósio Brasileiro de Engenharia de Software. p. 359-376. João Pessoa, PB, 2007. **(Qualis B3)**

[16] **A. S. Simão**, A. Petrenko, J. C. Maldonado. Comparing finite state machine test coverage criteria. IET Software, v. 3, p. 91-105, 2009. **(Qualis B2)**

[17] F. Dusse, **A. S. Simão**, J. C. Maldonado. Análise de Mutantes Aplicada a Critérios de Cobertura de Teste a partir de MEFs. In: The 3rd Brazilian Workshop on Systematic and Automated Software Testing. p. 41-50. Gramado, RS, 2009

ção de que critérios mais exigentes obtém também melhores escores de mutação e, por conseguinte, devem resultar em melhores casos de teste.

Resumo

Foram orientados quatro trabalhos de mestrado, três trabalhos de iniciação científica e um trabalho de conclusão de curso. Os trabalhos contaram com a colaboração ativa de diversos pesquisadores, em especial, Alexandre Petrenko, com quem realizou o trabalho de pós-doutoramento; José Carlos Maldonado, que foi o orientador do autor durante o mestrado e doutorado; Adilson Luiz Bonifácio, Arnaldo Moura e Simone do Rocio Senger de Souza. Foram publicados quatro artigos em revistas internacionais, seis artigos em eventos internacionais e sete artigos em eventos nacionais.

2.2 Teste de Programas Paralelos

Nesta seção, são descritos os trabalhos desenvolvidos pelo autor na área de pesquisa relacionada ao teste de programas paralelos. Foram desenvolvidos critérios, estratégias e ferramentas para o teste de programas paralelos.

O teste estrutural, ou teste caixa branca, utiliza a estrutura do programa para definir critérios de teste. Um critério de teste estabelece requisitos que um conjunto de teste deve atender, servindo tanto para avaliar a adequação de um conjunto quanto para guiar a geração de conjuntos adequados. Os critérios de teste são algumas vezes chamados de critérios de cobertura, pois em geral exige-se que elementos específicos do programa sejam "cobertos", ou seja, sejam executados sob determinadas condições. Em geral, a estrutura do programa é abstraída na forma de um grafo de fluxo de controle (GFC), no qual cada nó representa um bloco de comandos sem desvio de controle (ou seja, ou todos os comandos de um bloco são executados, ou nenhum o é), e cada aresta representa o desvio de controle entre dois blocos.

Critérios de teste para programas paralelos foram definido em (Souza et al., 2008)[18]. Primeiramente, o conceito de grafos de fluxo de controle foi estendido de modo a ser aplicado para programas paralelos. Um programa paralelo foi modelado com um conjunto de processos concorrentes, que se comunicam pela troca de mensagens. Foram incluídos arestas de sincronização/comunicação, que representam o envio de uma mensagem de um processo a outro, definindo um GFC paralelo. Mais especificamente, se o nó n_1 do GFC de um processo possui um comando que envia uma mensagem que pode ser recebida por um comando de um n_2 do GFC de outro processo, uma aresta inter-processo entre os nós n_1 e n_2 é criada. Então, foram definidos critérios de cobertura que levam em consideração tais arestas.

Os critérios propostos complementam os critérios estruturais introduzidos por (Rapps and Weyuker, 1985) para programas sequenciais. Foi desenvolvida a ferramenta ValiPar, que apoia o teste estrutural de programas paralelos baseados nos critérios definidos (Souza et al., 2005)[19]. A ferramenta foi desenvolvida em módulos, como descrito na Figura 2.3.

Os módulos que compõem a ferramenta são:

Vali-Inst: É responsável pela geração do modelo de teste, instrumentação e extração das informações de fluxo de dados. A geração do modelo de teste cria uma representação do programa de entrada no modelo GFC paralelo. A ferramenta ValiPar utiliza uma abordagem conservativa para gerar os arcos inter-processos. A instrumentação gera um programa instrumentado em que comandos são inseridos no programa original para gravar informações sobre trechos executados. A extração das informações de fluxo de dados armazena informações sobre definições e usos de variáveis.

---

[18] S. R. S. Souza, S. R. Vergílio, P. S. L. Souza, **A. S. Simão**, A. Hausen. Structural Testing Criteria for Message-Passing Parallel Programs. Concurrency and Computation. Practice & Experience, v. 20, p. 1893-1916, 2008. **(Qualis B1)**

[19] S. R. S. Souza, S. R. Vergílio, P. S. L. Souza, **A. S. Simão**, T. B. Gonçalves, A. M. Lima, A. C. Hausen. ValiPar: A Testing Tool for Message-Passing Parallel Programs. In: XVII International Conference on Software Engineering and Knowledge Engineering. p. 386-392, Taipen, Taiwan, 2005. **(Qualis B2)**

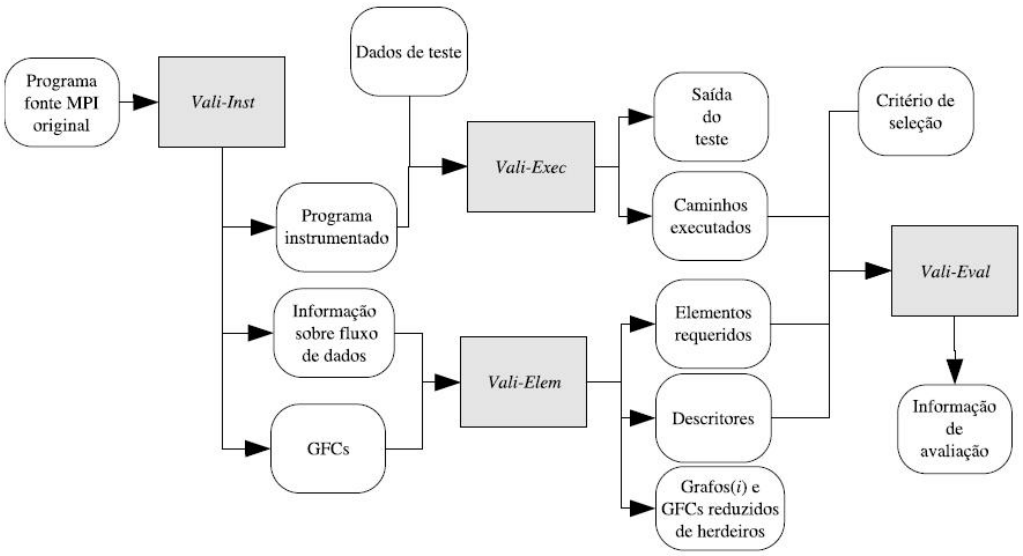


Figura 2.3: Arquitetura da Ferramenta ValiPar.

Vali-Exec: É responsável pela execução controlada do programa instrumentado e gravação das instruções ocorridas. O programa é executado com os casos de teste fornecidos pelo usuário. As saídas dos testes e os *traces* são armazenados para realização da análise de cobertura.

Vali-Elem: É responsável pela geração dos elementos requeridos dos critérios de teste do modelo GFC paralelo. Esses elementos são gerados por meio do modelo de teste e informações de fluxo de dados fornecidos pelo módulo Vali-Inst.

Vali-Eval: É responsável pela avaliação da cobertura dos casos de teste em relação aos critérios de teste selecionados. Utiliza informações dos módulos Vali-Elem e Vali-Exec para determinar qual foi a cobertura alcançada pelos casos de teste executados.

A ferramenta pode ser instanciada para diversas plataformas de computação paralela. A ValiMPI é a versão para *Message Passing Interface* (MPI) da ValiPar

(Hausen et al., 2006)[20], (Hausen et al., 2007)[21]. Foi também desenvolvida uma versão para Parallel Virtual Machine (PVM).

Os critérios definidos para a passagem de mensagem foram estendidos para aplicar em outro paradigma de programação paralelo, quando a comunicação inter-processos é realizada por meio de memória compartilhada (Sarmanho et al., 2007)[22]. O principal problema é como as várias linhas de execução compartilham as variáveis e como isso impacta no fluxo de dados do programa. A sincronização entre as linhas é realizadas por meio de semáforos (Sarmanho et al., 2008)[23].

Como os critérios são baseados em sincronizações em geral produzem muitos elementos requeridos, 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. No trabalho de iniciação científica de Mário dos Santos Camillo, foi investigado como a estratégia proposta por (Lei and Carver, 2006) poderia ser utilizada para gerar somente as sincronizações executáveis.

O modelo baseado em GFC para programas paralelos mostrou-se bastante versátil para representar sistemas no contexto de composição de serviços web. Um serviço web é um componente autônomo de software que pode ser invocado por meio de protocolos abertos. Dois ou mais serviços web podem ser combinados para formar um novo serviço web, em um processo conhecido como composição de serviços web. Existem duas formas principais de composição: coreografia ou orquestração. No caso de orquestração, um serviço web princi-

[20] A. C. Hausen, S. R. Vergílio, S. R. S. Souza, P. S. L. Souza, **A. S. Simão**. ValiMPI: Uma Ferramenta para o Teste de Programas Paralelos. In: Sessão de Ferramentas - Simpósio Brasileiro de Engenharia de Software. p. 1-6, Florianópolis, SC, 2006.

[21] 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: The 8th IEEE LAtin-American Test Workshop, p1-6 Cuzco, Peru. 2007. **(Qualis B3)**

[22] F. S. Sarmanho, P. S. L. Souza, S. R. S. Souza, **A. S. Simão**. Aplicação de Teste Estrutural para Programas Multithreads Baseados em Semáforos. In: The 1st Workshop on Languages and Tools for Parallel and Distributed Programming (LTPD). p. 18-21, Granado, RS, 2007.

[23] F. S. Sarmanho, P. S. L. Souza and S. R. S. Souza, **A. S. Simão**. Structural Testing for Semaphore-Based Multithread Programs. In: The 8th International Conference on Computational Science. p. 337-346, Kraków, Polônia, 2008.

pal faz chamadas síncronas e assíncronas a outros serviços. Dessa forma, uma composição de serviços web pode ser encarada como um sistema paralelo. Essa perspectiva foi explorada no trabalho de mestrado de Andre Takeshi Endo (Endo et al., 2007)[24] e (Endo et al., 2008)[25].

Observou-se que em algumas situações o modelo baseado em GFC não é suficiente para representar adequadamente todos os aspectos de uma composição de serviços. Assim, em (Endo et al., 2010)[26] 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., 2006).

Resumo

Os trabalhos nessa linha contaram com a colaboração ativa dos alunos de mestrado e iniciação científica, e também com diversos pesquisadores, em especial, Simone do Rocio Senger de Souza, Paulo Sérgio Lopes de Souza e Silvia R. Vergílio. Foram orientados um trabalho de mestrado e um trabalho de iniciação científica. Foram publicados um artigo em revista internacional, seis artigos em eventos internacionais e dois artigos em eventos nacionais.

2.3 Outras Linhas

Nesta seção, são apresentadas as contribuições que não se encaixam nas linhas principal e secundária apresentadas nas seções anteriores. De modo geral, tratam-se de linhas que ainda estão se desenvolvendo ou que representam uma

[24] A. T. Endo, **A. S. Simão**, S. R. S. Souza, P. S. L. Souza. Aplicação de Teste Estrutural para Composição de Web Services. In: The 1st Brazilian Workshop on Systematic and Automated Software Testing. p. 13-20, João Pessoa, PB, 2007.

[25] A. T. Endo, **A. S. Simão**, S. R. S. Souza and P. S. L. Souza. Web Services Composition Testing: A Strategy Based on Structural Testing of Parallel Programs. In: TaicPart: Testing Academic & Industrial Conference - Practice and Research Techniques. p. 3-12, Windsor, UK, 2008.

[26] A. T. Endo, M. Lindshulte, **A. S. Simão**, S. R. S. Souza. Event- and Coverage-Based Testing of Web Services. In: 2nd Workshop on Model-Based Verification & Validation From Research to Practice (MVV). p. 1-8, Cingapura, 2010.

colaboração pontual. Contudo, são linhas que podem vir a se desenvolver futuramente.

2.3.1 Teste de Mutação

O teste de mutação é uma técnica de teste baseado em defeitos, ou seja, utiliza-se o conhecimento de erros típicos cometidos pelos desenvolvedores para guiar a adequação de conjuntos de casos de teste. Durante o mestrado e o doutorado, o autor investigou a aplicabilidade do teste de mutação para a geração de testes a partir de especificações, mais precisamente, de Redes de Petri e Redes de Petri Coloridas. Dessa forma, ao término do doutorado, algumas contribuições referentes a essa ainda foram obtidas, refletindo os resultados de investigações iniciadas no doutorado e concluídas posteriormente.

O teste de mutação envolve uma série de passos que devem ser seguidos para que o resultado obtido seja de qualidade. Cada passo por si só apresenta desafios interessantes que têm sido atacados por meio de diversas contribuições teóricas e práticas na área. Em uma tentativa de sistematizar essa contribuições, ou seja, de catalogar e organizar as contribuições obtidas, em (Vincenzi et al., 2005)[27] é proposto um processo de teste baseado em mutação. O processo, cuja versão estendida é apresentada em (Vincenzi et al., 2006)[28], descreve todos os passos referentes ao teste de mutação e como os problemas encontrados podem ser resolvidos com diversas contribuições encontradas na literatura.

A qualidade do teste de mutação está diretamente relacionada à qualidade dos mutantes utilizados. A primeira tarefa a ser realizada quando o teste de mutação vai ser aplicado em um novo contexto, tais como uma nova linguagem de programação ou técnica de especificação, é definir um conjunto de operadores de mutação. Um operador de mutação é uma função que, dado o artefato original,

---

[27] A. M. R. Vincenzi, M. E. Delamaro, **A. S. Simão**, J. C. Maldonado. Muta-Pro: Towards the Definition of a Mutation Testing Process. In: The 6th LAtin-American Test Workshop. p. 149-154, Salvador, BA, 2005. **(Qualis B3)**

[28] A. M. R. Vincenzi, M. E. Delamaro, **A. S. Simão**, J. C. Maldonado. Muta-Pro: Towards the Definition of a Mutation Testing Process. Journal of the Brazilian Computer Society, v. 12, p. 47-61, 2006. **(Qualis B2)**

produz um conjunto de artefatos, cada um com alguma modificação refletindo um possível engano que pode ser cometido pelo desenvolvedor. Os operadores de mutação são, portanto, de suma importância para definir a qualidade do teste baseado em mutação. Em (Simão et al., 2009a)[29], foi proposta uma linguagem, chamada $\mathcal{MuDeL}$, que utiliza os conceitos dos paradigmas transformacional (em especial, a linguagem TXL (Cordy et al., 1988)) e lógico (em especial, a linguagem Prolog (Bratko, 1990)) para definir operadores de mutação. Com base na gramática livre de contexto da linguagem alvo, são criados diversos módulos que manipulam um artefato produzido nessa linguagem. Inicialmente, a árvore sintática é obtida. Em seguida, com base nos comandos do operador de mutação escritos em $\mathcal{MuDeL}$, a árvore sintática é alterada, dando origem a diversas árvores mutantes. Por fim, os nós da árvore são visitados de modo a obter os mutantes. Todo o processo é automatizado, de forma que o desenvolvedor necessita apenas definir a gramática livre de contexto (geralmente disponível para diversas linguagens de programação) e depois os operadores propriamente ditos. A linguagem foi definida de forma a estimular o reúso dos operadores entre linguagens similares.

2.3.2 Teste de Regressão

Em geral, um software é desenvolvido por meio de várias versões, de forma que uma versão introduz novas funcionalidades ou corrige problemas em versões anteriores. Após a criação de uma nova versão, é importante se certificar que problemas indesejáveis não tenham sido introduzidos no software. Por exemplo, ao se tentar corrigir um problema em alguma parte do software, defeitos podem ser incluídos em outras partes. Para evitar que isso ocorra, o teste de regressão é geralmente aplicado. O teste de regressão busca aplicar testes para garantir que defeitos não tenham sido inadvertidamente introduzidos no software. Os testes utilizados durante o desenvolvimento das versões anteriores são em geral aplicados. Contudo, a reexecução de todos os testes pode ser custosa. Diversas

[29] **A. S. Simão**, J. C. Maldonado and R. S. Bigonha. A transformational language for mutant description. Computer Languages, Systems & Structures, v. 35, p. 322-339, 2009. **(Qualis B2)**

técnicas de priorização e seleção de testes para otimizar o teste de regressão tem sido propostos (Rothermel et al., 2001).

Em (Simão et al., 2006)[30], foi proposta uma estratégia para a seleção de casos de teste para o teste de regressão que é baseado em redes neurais. Para cada caso de teste, é extraída uma assinatura que representa a execução do caso de teste. A assinatura foi definida como sendo a quantidade de vezes que um nó foi exercitado durante a execução. Em seguida, a rede neural particiona os casos de teste em blocos que continham casos de teste com assinaturas semelhantes. Um caso de teste de cada bloco é então selecionado. A premissa é que casos de teste com assinaturas semelhantes exercitam características semelhantes do software e, portanto, durante o teste de regressão, deve-se priorizar os testes mais distintos possíveis.

Em (Simão et al., 2008)[31], a abordagem foi aplicada utilizando-se como assinatura os pares de definição e uso de variáveis. A abordagem foi comparada com as abordagens *Testar-Tudo* e *Aleatória*, conseguindo uma economia de tempo de execução em relação à primeira e na quantidade de defeitos encontrados em relação à segunda.

2.3.3 Teste Funcional

O teste funcional utiliza as informações referentes a especificação do software para avaliar a adequação de um conjunto de casos de teste. Também pode ser utilizado como guia para a geração de casos de teste. Contudo, enquanto o teste estrutural em geral possui um grande apoio ferramental, o teste funcional é normalmente aplicado manualmente, tanto para a geração quanto para a avaliação

[30] **A. S. Simão**, R. F. Mello, L. J. Senger. A Technique to Reduce the Test Case Suites for Regression Testing Based on a Self-Organizing Neural Network Architecture. In: The 30th Annual International Computer Software and Applications Conference. p. 1-4, Chicago, USA, 2006. **(Qualis B1)**

[31] **A. S. Simão**, R. F. Mello, L. J. Senger, L. T. Yang, Improving regression testing performance using the Adaptive Resonance Theory-2A self-organising neural network architecture. International Journal of Autonomous and Adaptive Communications Systems, pp. 370-385, 2008.

da adequação dos casos de teste. Em (Rocha et al., 2005)[32], foi proposta uma ferramenta baseada em aspectos com o objetivo de automatizar a avaliação de conjuntos de casos de teste com base na técnica funcional. Foram implementados os critérios *Particionamento em Classes de Equivalência* e *Análise de Valor Limite*. Para cada condição de entrada, é definido uma classe de forma que cada classe de equivalência é representada por um método. O método, que retorna um valor *booleano*, decide se um conjunto de parâmetros de entrada de uma operação estão em uma classe de equivalência. São definidos então aspectos que interceptam as chamadas das operações do sistema e enfocam os métodos que correspondem às classes. Assim, pode-se determinar quais classes foram cobertas.

2.4 Considerações Finais

Na Figura 2.4, são apresentadas as publicações obtidas após a conclusão do doutorado, separadas por ano e por linha de pesquisa. A numeração corresponde ao número da nota de rodapé utilizada para introduzir a referência no decorrer deste Capítulo 2. Como mencionado anteriormente, os círculos correspondem às publicações em revista, enquanto que os quadrados correspondem às publicações em conferências. As publicações nacionais são apresentadas com linha tracejada. As oito publicações mais relevantes são apresentadas com fundo cinza; cópias dessas publicações podem ser encontradas nos Apêndices C a J.

Pode-se observar que o volume de publicações se manteve adequado ao longo de todo o período. Inicialmente, os trabalhos estavam mais dispersos entre as diversas linhas de pesquisa, sendo que a maioria dos trabalhos eram publicados em conferências nacionais. O perfil se altera a partir de 2008, coincidindo com o período de pós-doutorado; mais publicações em conferências internacionais e em revistas foram obtidas, concentrando-se principalmente no teste baseado em MEFs, tema do pós-doutorado.

[32] A. D. Rocha, **A. S. Simão**, J. C. Maldonado, P. C. Masiero. Uma ferramenta baseada em aspectos para o teste funcional de programas Java. In: Simpósio Brasileiro de Engenharia de Software. p. 263-278, Uberlandia, MG, 2005. **(Qualis B3)**

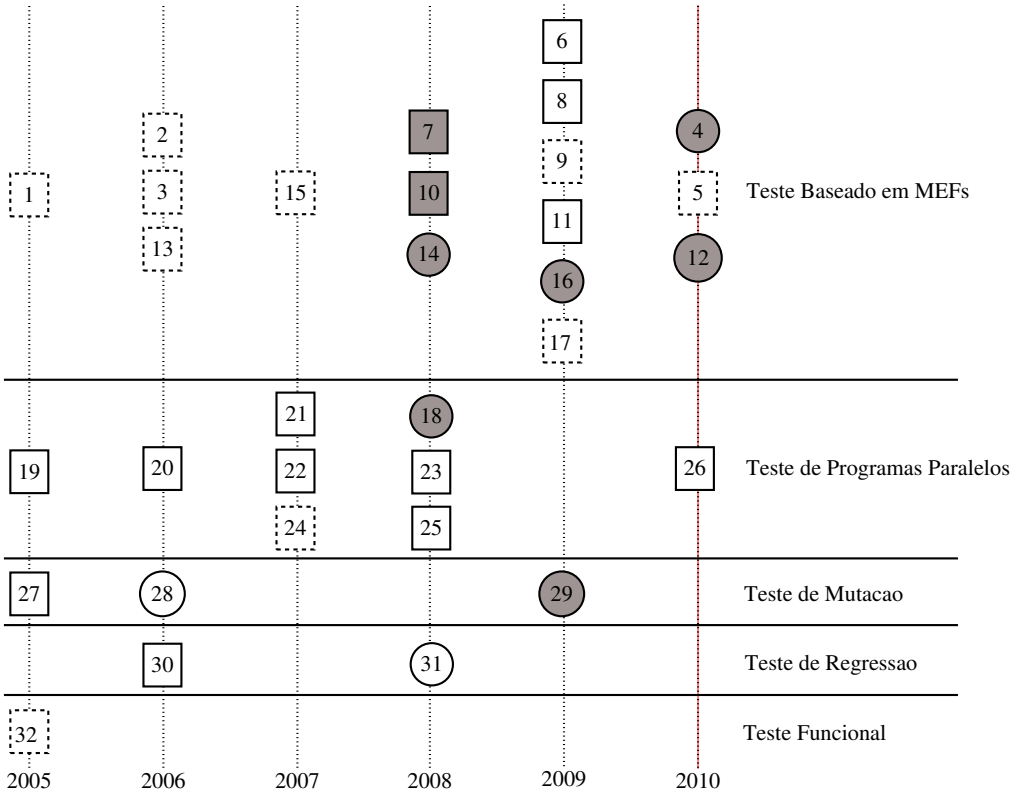


Figura 2.4: Distribuição das Publicações.

Outro ponto a se destacar é que o autor atuou nas três técnicas de teste, a saber, estrutural (no teste de programas paralelos), funcional (incluindo o teste baseado em modelos) e baseada em defeitos (no teste de mutação). Portanto, obteve-se uma visão ampla da área de pesquisa. Vale lembrar que apesar de estar dividido em cinco linhas distintas de pesquisa, os trabalhos estão todos relacionadas ao teste de software, que é uma subárea da engenharia de software.

As contribuições para a área de teste de software podem ser classificadas em: estudos teóricos, estudos experimentais e automatização. Na Figura 2.5, é apresentado como as publicações obtidas podem ser mapeadas nessas três categorias. Pode-se observar que o autor tem atuado nas três categorias, com maior destaque nos estudos teóricos. Em geral, estudos teóricos podem abrir novas linhas de investigação nas duas outras categorias. Assim, espera-se que futuramente trabalhos envolvendo estudos experimentais e automatização possam ser desenvolvidos.

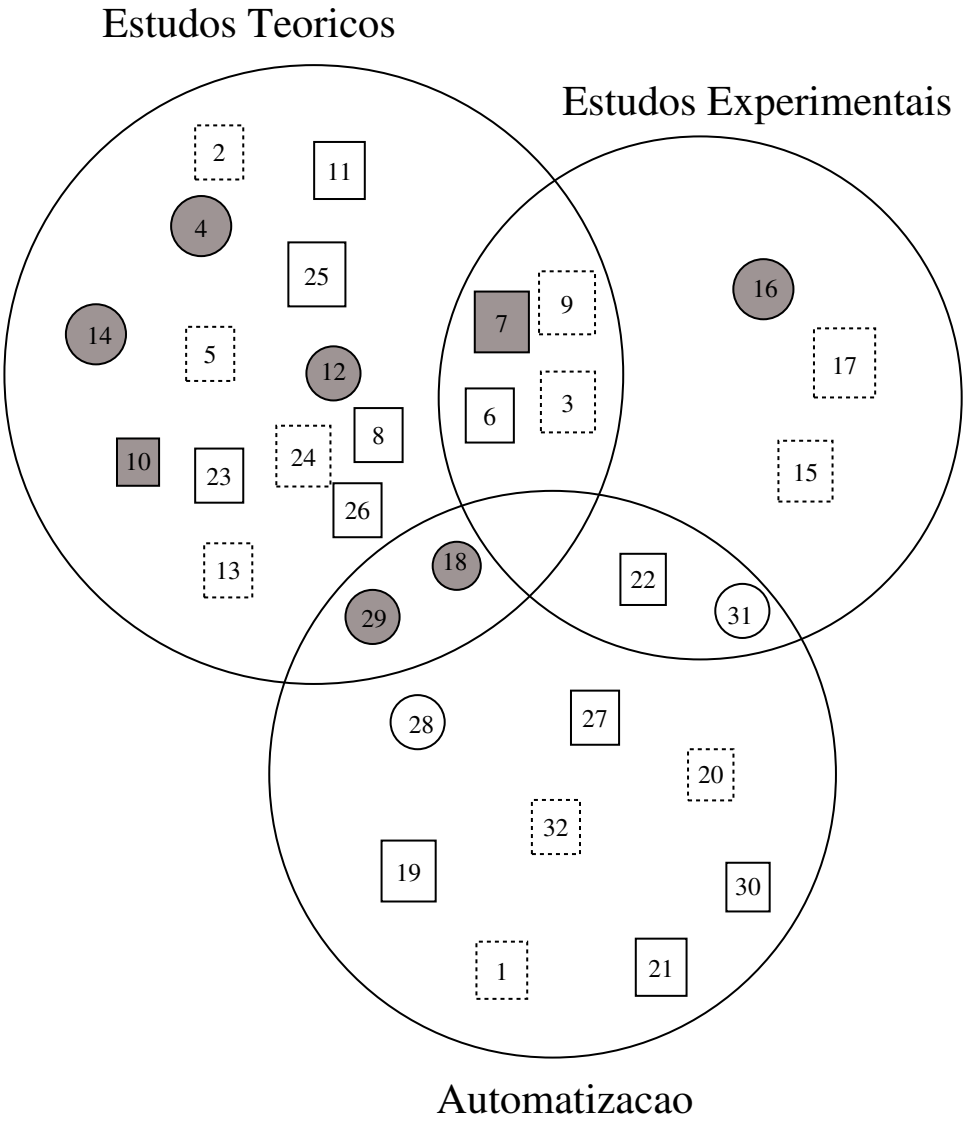


Figura 2.5: Distribuição das Publicações: Contribuições.

Capítulo 3

Conclusões

3.1 Discussão e Reflexões

Após a conclusão do doutorado, o autor continuou a desenvolver pesquisas em teste de software. Várias linhas de pesquisa dentro dessa área foram investigadas. Maior ênfase foi dada ao teste baseado em MEFs, que caracteriza a linha principal de pesquisa. Diversos resultados relevantes foram obtidos, com avanços em importantes problemas que estão sendo investigados há algumas décadas. Tratam-se de problemas fundamentais que foram alvo de diversos trabalhos ao longo do tempo.

Outra linha de investigação que foi explorada pelo autor é o teste de programas paralelos. Assim como novos desafios são adicionados à programação de programas paralelos, o teste de programas paralelos também difere do teste de programas tradicionais. Pode-se observar que as publicações nessa linha são anteriores a 2009. De fato, em agosto de 2008, o autor afastou-se para realizar pós-doutorado (a propósito, para dar continuidade às investigações na linha do teste baseado em MEFs). Dessa forma, houve uma interrupção natural no desenvolvimento dessa linha por parte do autor. Espera-se que no futuro essa linha volte a ser investigada.

Um ponto a se destacar é que apenas os trabalhos desenvolvidos após a conclusão do doutorado foram apresentados neste documento. Assim, não foram

incluídos diversos trabalhos desenvolvidos e publicados antes da conclusão, o que se constitui de 3 publicações em revistas e 13 trabalhos em conferências. O tema principal desses trabalhos é o teste de mutação, em especial no teste de especificações.

Atualmente, o autor possui uma boa inserção no cenário acadêmico. O autor é membro do comitê de programa do Simpósio Brasileiro de Engenharia de Software, do Simpósio Brasileiro de Métodos Formais e do Simpósio Brasileiro de Qualidade de Software, que se constituem nos principais eventos nacionais na sua área de pesquisa. Foi também *co-chair* de um evento internacional (22nd IFIP International Conference on Testing Software and Systems — ICTSS) e *co-chair* do primeiro workshop brasileiro voltado para a área de teste de software (Brazilian Workshop On Systematic and Automated Software Testing — SAST). O autor é também *co-chair* do Simpósio Brasileiro de Métodos Formais 2011.

3.2 Trabalhos Futuros e em Andamento

Atualmente, o autor orienta um aluno de doutorado e um aluno de mestrado, ambos com temas relacionados ao teste baseado em MEFs. Note-se que, por estar afastado para a realização de pós-doutorado, o número de alunos que puderam ser orientados foi reduzido; porém, deve-se nos próximos anos aumentar o número de trabalhos.

No trabalho de doutorado do aluno André Endo, investiga-se métodos de geração de teste para serviços Web, dando continuidade ao trabalho desenvolvido por ele no mestrado. Os métodos de geração de teste a partir de MEFs estão sendo estudados, de forma a identificar quando e como eles podem ser aplicados nesse contexto. De certa forma, pode-se dizer que esse trabalho une as duas principais linhas de pesquisa do autor, uma vez que serviços web podem ser tratados como programas paralelos e os métodos de geração estudados foram resultados dos trabalhos com MEFs.

No trabalho de mestrado da aluna Arineiza Cristina Pinheiro, investiga-se como os métodos podem ser aplicados no teste de sistemas embarcados. O interesse nesse contexto de aplicação deve-se ao fato da participação do autor no Ins-

tituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos (INCT-SEC). Um dos objetivos do trabalho de Pinheiro é identificar uma aplicação real que possa ser testada com os métodos de geração baseados em MEFs. Eventualmente, a ferramenta Plavis/FSM poderá ser reestruturada para se adequar a esse contexto.

Além disso, foi recentemente aprovado um projeto de pesquisa pelo Edital Universal do CNPq, que tem como tema a continuidade dos trabalhos de investigação sobre métodos de geração de conjuntos completos para MEFs. Dessa forma, pode-se destacar como trabalhos futuros nessa linhas o estudo dos seguintes itens:

- Geração de testes para MEFs não determinísticas; todos os trabalhos desenvolvidos pelo autor até o momento tratam de MEFs determinística. Contudo, existem domínios que podem ser melhores descritos com a inclusão de não determinismo.

- Geração de sequências de verificação a partir de MEFs sem sequências de distinção.

- Consolidação de aplicações práticas de teste baseado em MEFs para domínios específicos, tais como sistemas embarcados e arquiteturas orientadas a serviço.

Agradecimentos

Os pesquisadores e alunos aqui citados tiveram participação fundamental nas atividades de pesquisas desenvolvidas. Em especial, agradeço aos professores do ICMC José Carlos Maldonado, Simone do Rocio Senger de Souza e Paulo Sérgio Lopes de Souza, e ao pesquisador Alexandre Petrenko.

Agradeço aos professores e funcionários do ICMC, muitos dos quais me apoiam desde que eu ainda era aluno de pós-graduação.

Agradeço ao suporte financeiro da FAPESP, CAPES e CNPq.

Agradeço aos meus pais, Geraldo e Olivia, por sempre me apoiarem.

Finalmente, agradeço a Selma, Emanoela e Gabriel, pelo carinho, apoio e compreensão.

Referências Bibliográficas

Aho, A. V., Dahbura, A. T., Lee, D., and Uyar, M. U. (1991). An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615.

Alberto, A. and Simão, A. (2009). Minimization of incompletely specified finite state machines based on distinction graphs. In *Latin-American Test Workshop*, pages 1–6, Buzios, RJ.

Andrews, A., Offutt, J., and Alexander, R. (2005). Testing web applications by modeling with fsms. *Software Systems and Modeling*, 4(3):326–345.

Beizer, B. (1990). *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

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

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley Longman, Inc., 1 edition.

Bonifácio, A., Moura, A., and Simão, A. (2008a). A generalized model-based test generation method. In *Proceedings of 6th IEEE International Conferences on Software Engineering and Formal Methods*, pages 139–148, Cape Town, Africa do Sul.

Bonifácio, A. L., Moura, A., Simão, A., and Maldonado, J. C. (2008b). Towards deriving test sequences by model checking. *Electronic Notes in Theoretical Computer Science*, 195:21–40.

Bonifácio, A. L., Simão, A., Moura, A., and Maldonado, J. C. (2006). Conformance testing by model checking timed extended finite state machines. In *Simpósio Brasileiro de Métodos Formais*, pages 43–58, Natal, RN.

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Wokingham, England ; Reading, Mass., 2 edition.

Candolo, M. A., Simao, A. S., and Maldonado, J. C. (2001). Mgaset - uma ferramenta para apoiar o teste e validação de especificações baseadas em máquinas de estado finito. In *Anais do XV Simpósio Brasileiro de Engenharia de Software*, pages 386–391.

Chen, J., Hierons, R. M., Ural, H., and Yenigun, H. (2005). Eliminating redundant tests in a checking sequence. In *TestCom 2005*, number 3502 in Lecture Notes on Computer Science, pages 146–158.

Chow, T. S. (1978). Testing software design modeled by finite-state-machines. *IEEE Transactions on Software Engineering*, 4(3):178–186.

Cordy, J. R., Halpen, C. D., and Promislow, E. (1988). TXL: A rapid prototyping system for programming language dialects. In *IEEE International Conference on Computer Languages*, Maimi.

Cutigi, J. F., Ribeiro, P. H., Simão, A., and Souza, S. R. S. (2010). Redução do número de seqüências no teste de conformidade de protocolos. In *XI Workshop de Testes e Tolerância a Falhas*, volume 1, pages 105–117, Gramado, RS.

Davis, A. M. (1988). A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, 31(9).

de Mello Neto, L. F. (2008). Minimização de conjuntos de casos de teste para máquinas de estados finitos.

DeMillo, R. A. (1980). Mutation analysis as a tool for software quality assurance.

Dorofeeva, R., El-Fakih, K., and Yevtushenko, N. (2005a). An improved conformance testing method. In *FORTE*, pages 204–218.

Dorofeeva, R., Yevtushenko, N., El-Fakih, K., and Cavalli, A. R. (2005b). Experimental evaluation of fsm-based testing methods. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 23–32, Washington, DC, USA. IEEE Computer Society.

Dusse, F., Simão, A., and Maldonado, J. C. (2009). Análise de mutantes aplicada a critérios de cobertura de teste a partir de mefs. In *III Brazilian Workshop on Systematic and Automated Software Testing*, pages 41–50, Gramado, RS.

Endo, A. T. (2008). Teste de composição de web services: uma estratégia baseada em um modelo de teste de programas paralelos.

Endo, A. T., Lindshulte, M., Simão, A., and Souza, S. R. S. (2010). Event- and coverage-based testing of web services. In *2nd Workshop on Model-Based Verification & Validation From Research to Practice*, pages 1–8, Cingapura, Cingapura.

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

Endo, A. T., Simão, A., Souza, S. R. S., and Souza, P. S. L. (2007). Aplicação de teste estrutural para composição de web services. In *Brazilian Workshop on Systematic and Automated Software Testing*, pages 13–20, João Pessoa, PB.

Fabbri, S. C. P. F. and Maldonado, J. C. (2001). Teste de software. In Rocha, A. R. C., Maldonado, J. C., and Weber, K. C., editors, *Qualidade de Software: Teoria e Prática*, chapter 4, pages 73–84. Prentice-Hall, São Paulo, Brasil.

Fabbri, S. C. P. F., Maldonado, J. C., Delamaro, M. E., and Masiero, P. C. (1999). Proteum/FSM: A tool to support finite state machine validation based on mu-

tation testing. In *XIX SCCC - International Conference of the Chilean Computer Science Society*, pages 96–104, Talca, Chile.

Fujiwara, S., Bochman, G. V., Khendek, F., Amalou, M., and Ghedamsi, A. (1991). Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603.

Gill, A. (1962). *Introduction to the Theory of Finite-State Machines*. McGraw-Hill, New York.

Gonenc, G. (1970). A method for design of fault detection experiments. *IEEE Transactions on Computers*, 19(6):551–558.

Harrold, M. J. (2000). Testing: A roadmap. In *In The Future of Software Engineering*, pages 61–72. ACM Press.

Hausen, A. C., Vergilio, S. R., Souza, S. R. S., Souza, P. S. L., and Simão, A. (2007). A tool for structural testing of mpi programs. In *LAtin-American Test Workshop - LATW*, pages 1–6, Cuzco, Peru.

Hausen, A. C., Vergílio, S. R., Souza, S. R. S., Souza, P. S. L., and Simão, A. (2006). Valimpi: Uma ferramenta para o teste de programas paralelos. In *Sessão de Ferramentas - Simpósio Brasileiro de Engenharia de Software, 2006*, pages 1–6, Florianópolis, SC.

Hennie, F. C. (1964). Fault-detecting experiments for sequential circuits. In *Proceedings of Fifth Annual Symposium on Circuit Theory and Logical Design*, pages 95–110.

Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A. J. H., Vilkomir, S., Woodward, M. R., and Zedan, H. (2009). Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):1–76.

Hierons, R. M. and Ural, H. (2002). Reduced length checking sequences. *IEEE Transactions on Computers*, 51(9):1111–1117.

Hierons, R. M. and Ural, H. (2006). Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55(5):618–629.

Holzmann, G. J. (1991). *Design and Validation of Protocols*. Prentice-Hall Software Series, Englewood Cliffs, New Jersey.

Hong, H. S., Kwon, Y. R., and Cha, S. D. (1995). Testing of object-oriented programs based on finite state machines. In *2nd Asia-Pacific Software Engineering Conference (APSEC'95)*, pages 234–241, Brisbane, Queensland, Australia. IEEE Computer Society.

Lei, Y. and Carver, R. H. (2006). Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32:382–403.

Luo, G., Petrenko, R., and Bochmann, G. V. (1994). Selecting test sequences for partially-specified nondeterministic finite state machines. In *In IFIP 7th International Workshop on Protocol Test Systems*, pages 91–106.

Maldonado, J. C. (1991). *Critério potenciais usos: Uma contribuição ao teste estrutural de software*. PhD thesis, DCA/FEE/UNICAMP, Campinas.

Maldonado, J. C., Barbosa, E. F., Vincenzi, A. M. R., Delamaro, M. E., Souza, S. R. S., and Jino, M. (2004). Introdução ao teste de software. Technical Report 65, ICMC/USP, São Carlos, SP. Notas Didáticas do ICMC, Série Computação.

Martins, E., Selma B. Sabi a., and Ambrosio, A. M. (1999). Condata: A tool for automating specification-based test case generation for communication systems. *Software Quality Control*, 8(4):303–320.

Mello Neto, L. F. and Simão, A. (2007). Minimização de conjuntos de casos de teste por meio de condições de suficiência. In *1st Brazilian Workshop on Systematic and Automated Software Testing*, pages 55–62, João Pessoa, PB.

Mello Neto, L. F. and Simão, A. (2008). Test suite minimization based on fsm completeness sufficient conditions. In *Proceedings of 9th IEEE Latin-American Test Workshop*, pages 93–98, Puebla, Mexico.

Moore, E. F. (1956). Gedanken-experiments on sequential machines. (34):129–153.

Myers, G. J., Badgett, T., and Thomas, T. M. (2004). *The Art of Software Testing*. Wiley, New York.

Naito, S. and Tsunoyama, M. (1981). Fault detection for sequential machines by transition tours. In *Proceedings of the 11th IEEE Fault Tolerant Computing Conference (FTCS 1981)*, pages 238–243. IEEE Computer Society Press.

Petrenko, A., von Bochmann, G., and Yao, M. Y. (1996). On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*, 29(1):81–106.

Petrenko, A. and Yevtushenko, N. (2005). Testing from partial deterministic fsm specifications. *IEEE Transactions on Computers*, 54(9):1154–1165.

Petrenko, A., Yevtushenko, N., Lebedev, A., and Das, A. (1993). Nondeterministic state machines in protocol conformance testing. In *Protocol Test Systems*, pages 363–378.

Pressman, R. S. (2005). *Engenharia de Software*. Makron Books do Brasil.

Pretschner, A. and Philipps, J. (2004). Methodological issues in model-based testing. In *Model-Based Testing of Reactive Systems*, Lecture Notes in Computer Science, pages 281–291.

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

Ribeiro, P. H., Cutigi, J. F., and Simão, A. (2009). Geração de seqüências de verificação baseada em algoritmos genéticos. In *III Workshop Brasileiro de Teste de Software Sistemático e Automatizado*, volume 1, pages 61–70, Gramado, RS.

Rocha, A. D., Simão, A., Maldonado, J. C., and Masiero, P. C. (2005). Uma ferramenta baseada em aspectos para o teste funcional de programas java. In *19o Simposio Brasileiro de Engenharia de Software*, pages 263–278, Uberlandia, MG.

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transaction on Software Engineering*, 27(10):929–948.

Sabnani, K. K. and Dahbura, A. (1988). A protocol test generation procedure. *Computer NetWorks and ISDN Systems*, 15(4):285–297.

Sarmanho, F. S., Souza, P. S. L., Souza, S. R. S., and Simão, A. (2007). Aplicação de teste estrutural para programas multithreads baseados em semáforos. In *1st Workshop on Languages and Tools for Parallel and Distributed Programing (LTPD)*, pages 18–21, Granado, RS.

Sarmanho, F. S., Souza, P. S. L., Souza, S. R. S., and Simão, A. (2008). Structural testing for semaphore-based multithread programs. In *Proceedings of International Conference on Computer Science*, pages 337–346, Kraków, Poland.

Sidhu, D. P. and Leung, T. K. (1989). Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426.

Simão, A. (2007). Teste baseado em modelos.

Simão, A., Ambrosio, A. M., Fabbri, S. C. P. F., Amaral, A. S., Martins, E., and Maldonado, J. C. (2005). Plavis/fsm: an environment to integrate fsm-based testing tools. In *Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software*, pages 1–6, Uberlândia, MG.

Simão, A., Maldonado, J. C., and Bigonha, R. S. (2009a). A transformational language for mutant description. *Computer Languages, Systems & Structures*, 35:322–339.

Simão, A., Mello, R. F., and Senger, L. J. (2006). A technique to reduce the test case suites for regression testing based on a self-organizing neural network architecture. In *30th Annual International Computer Software and Applications Conference*, pages 1–4, Chicago, USA.

Simão, A., Mello, R. F., Senger, L. J., and Yang, L. T. (2008). Improving regression testing performance using the adaptive resonance theory-2a self-organising

neural network architecture. *International Journal of Autonomous and Adaptive Communications Systems*, 1:370–385.

Simão, A. and Petrenko, A. (2008). Generating checking sequences for partial reduced finite state machines. In *Proceedings of 20th IFIP Int. Conference on Testing of Communicating Systems (TESTCOM)*, pages 153–168, Tokyo, Japão.

Simão, A. and Petrenko, A. (2009). Checking sequence generation using state distinguishing subsequences. In *5th Workshop on Advances in Model Based Testing*, pages 1–10, Denver, USA.

Simão, A. and Petrenko, A. (2010a). Checking completeness of tests for finite state machines. *IEEE Transactions on Computers*, 59:1023–1032.

Simão, A. and Petrenko, A. (2010b). Fault coverage-driven incremental test generation. *Computer Journal*, 53:1508–1522.

Simão, A., Petrenko, A., and Maldonado, J. C. (2007). Experimental evaluation of coverage criteria for fsm-based testing. In *Anais do Simpósio Brasileiro de Engenharia de Software*, pages 359–376, João Pessoa, PB.

Simão, A., Petrenko, A., and Maldonado, J. C. (2009b). Comparing finite state machine test coverage criteria. *IET Software*, 3:91–105.

Simão, A., Petrenko, A., and Yevtushenko, N. (2009c). Generating reduced tests for fsms with extra states. In *21st IFIP Int. Conference on Testing of Communicating Systems and the 9th Int. Workshop on Formal Approaches to Testing of Software*, pages 129–147, Eidenhoven, Holanda.

Souza, S. R. S., Vergílio, S. R., Souza, P. S. L., Simão, A., Gonçalves, T. B., Lima, A. M., and Hausen, A. C. (2005). Valipar: A testing tool for message-passing parallel programs. In *Proceedings of the XVII International Conference on Software Engineering and Knowledge Engineering*, pages 386–392, Taipen, Taiwan.

Souza, S. R. S., Vergílio, S. R., Souza, P. S. L., Simão, A., and Hausen, A. (2008). Structural testing criteria for message-passing parallel programs. *Concurrency and Computation. Practice & Experience*, 20:1893–1916.

Ural, H., Wu, X., and Zhang, F. (1997). On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99.

Ural, H. and Zhang, F. (2006). Reducing the lengths of checking sequences by overlapping. *Lecture Notes on Computer Science*, (3964):274–288.

Utting, M. and Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Utting, M., Pretschner, A., and Legeard, B. (2006). A taxonomy of model-based testing. Technical report, Hamilton, New Zealand.

Vincenzi, A. M. R., Delamaro, M. E., Simão, A., and Maldonado, J. C. (2005). Muta-pro: Towards the definition of a mutation testing process. In *6th Latin-american test workshop (LATW)*, pages 149–154, Salvador, BA.

Vincenzi, A. M. R., Delamaro, M. E., Simão, A., and Maldonado, J. C. (2006). Muta-pro: Towards the definition of a mutation testing process. *Journal of the Brazilian Computer Society*, 12:47–61.

Weyuker, E. J. (1996). Using failure cost information for testing and reliability assessment. *ACM Trans. Softw. Eng. Methodol.*, 5(2):87–98.

Yannakakis, M. and Lee, D. (1995). Testing finite state machines: Fault detection. *J. Computer and System Science*, 50(2):209–227.

Yao, M. Y., Petrenko, A., and von Bochmann, G. (1994). Fault coverage analysis in respect to an fsm specification. In *IEEE INFOCOM94*, pages 768–775, Toronto, Canadá.

Apêndice A

Teste baseado em Máquinas de Estados Finitos: Definições e Exemplos

Neste apêndice, são apresentados os principais conceitos do teste baseado em Máquinas de Estados Finitos, bem como as definições formais dos conceitos discutidos na Seção 2.1. Este apêndice é baseado no capítulo de livro (Simão, 2007), e no Capítulo 3 da dissertação de mestrado de de Mello Neto (2008).

A.1 Definições

Uma MEF A pode ser representada formalmente por uma tupla $(S, s_0, X, Y, D_A, \delta, \lambda)$, (Petrenko and Yevtushenko, 2005) , onde:

- S é um conjunto finito de estados, incluindo o estado inicial s_0;
- X é um conjunto finito de entradas;
- Y é um conjunto finito de saídas;
- $D_A \subseteq S \times X$ é um domínio da especificação;
- δ é uma função de transição, $\delta : D_A \rightarrow S$, e

- λ é uma função de saída, $\lambda : D_A \rightarrow Y$;

Dados um estado $s_i \in S$ e uma entrada $x \in X$, diz-se que (s_i, x) é uma transição definida se e somente se $(s_i, x) \in D_A$. Os estados s_i e $s_j = \delta(s_i, x)$ são chamados de estado *inicial* e estado *final* da transição, respectivamente.

Sejam $M = (S, s_0, X, Y, D_M, \delta, \lambda)$ e $I = (T, t_0, X, Y, D_I, \Delta, \Lambda)$ duas MEFs que representam uma especificação e uma implementação, respectivamente. Uma sequência de entrada $\alpha = x_1 x_2 \ldots x_k \in X^*$ é chamada de sequência de entrada definida para o estado $s_i \in S$ se existe uma sequência de transições (s_{i1}, x_1), onde s_{ij+1} é o estado final da transição (s_{ij}, x_j). A notação $\Omega_M(s_i)$ representa o conjunto de todas as sequências de entrada definidas no estado s_i da máquina M. Para uma sequência de entrada α e uma estrada x, tal que αx é definido no estado s_i, define-se que $\delta(s_i, \alpha x) = \delta(\delta(s_i, \alpha), x)$ e $\lambda(s_i, \alpha x) = \lambda(s_i, \alpha)\lambda(\delta(s_i, \alpha), x)$. Para a sequência vazia, denotada por ϵ, define-se que, para todo $s \in S$, $\delta(s, \epsilon) = s$ e $\lambda(s, \epsilon) = \epsilon$.

Dois estados $s_i, s_j \in M$ são compatíveis se, para todo $\alpha \in \Omega_M(s_i) \cap \Omega_M(s_j)$, tem-se que $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$. Caso contrário, os estados são *distinguíveis*. Formalmente, os estados s_i e s_j são *distinguíveis* se existe uma sequência de entrada $\gamma \in \Omega_M(s_i) \cap \Omega_M(s_j)$, chamada de sequência de separação (*separating sequence*), tal que $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$.

O estado s_i é *quasi-equivalente* ao estado s_j, se $\Omega_M(s_i) \supseteq \Omega_M(s_j)$ e $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$ para todo $\alpha \in \Omega_M(s_j)$. Em outras palavras, um estado s_i é *quasi-equivalente* a um estado s_j se para toda entrada definida em s_j, s_i produzir a mesma saída.

Dados os estados $s_i, s_j \in S$ e uma sequência $\alpha \in \Omega_M(s_i)$ tal que $\delta_M(s_i, \alpha) = s_j$, diz-se que α é uma sequência de transferência (*transfer sequence*) de s_i para s_j. Um conjunto *state cover* Q de uma MEF M com n estados é definido como um conjunto com n sequência . de transferência, incluindo a sequência vazia ϵ, que leva M a partir de seu estado inicial para cada um dos estados.

Um conjunto *transition cover* P é um conjunto de sequências de entrada em que para cada transição definida (s, x), existe uma sequência de entrada $\alpha \in \Omega_M(s_0)$,

tal que $\delta(s_0, \alpha) = s$ e $\alpha x \in P$. Ou seja, o conjunto P faz com que a máquina execute cada transição e que, em seguida, pare.

Uma sequência de distinção (*distinguishing sequence*) é uma sequência de entrada d em que a sequência de saída produzida pela MEF M, em resposta à entrada d, identifica o estado da máquina M, ou seja, para todo $s_i, s_j \in S, s_i \neq s_j, \lambda_M(s_i, d) \neq \lambda_M(s_j, d)$.

Uma sequência UIO (*unique input/output sequence*) de um estado s_j, denotado por $UIO(s_j)$ é uma sequência de entrada/saída única para esse estado, ou seja, para todo $s_i \in S$ $s_j, \lambda_M(s_i, UIO(s_j)) \neq \lambda_M(s_j, UIO(s_j))$. Dessa forma, com a aplicação da sequência UIO pode-se distinguir o estado s_j de qualquer outro estado, pois a saída produzida é específica (única) do estado s_j.

Um conjunto de caracterização (*characterization set*), frequentemente chamado de conjunto W, é um conjunto de sequências de entrada tal que, para dois estados quaisquer s_j e s_i, $i \neq j$, existe uma sequência $\beta \in W$ tal que $\lambda_M(s_j, \beta) \neq \lambda_M(s_i, \beta)$. Em outras palavras, o conjunto W é um conjunto de sequências de entrada que possui uma sequência que diferencia todo par de estados existentes em M.

Um conjunto $W_j \subseteq \Omega_M(s_j)$ de sequências de entrada definidas é chamado de identificador de estado (*state identifier*) ou conjunto de separação (*separating set*) do estado s_j se para qualquer outro estado s_i existe $\alpha \in W_j \cap \Omega_M(s_i)$ tal que $\lambda_M(s_j, \alpha) \neq \lambda_M(s_i, \alpha)$. Em outras palavras, o conjunto W_j é um identificador do estado s_j se possui uma sequência de entrada α que o diferencia de todos os demais estados.

Uma família de separação (*separating family*) ou identificadores harmonizados (*harmonized identifiers*) é um conjunto de identificadores de estado $H_j, s_j \in S$, tal que para dois estados quaisquer $s_j,\ s_i \in S$, $i \neq j$, existe $\beta \in H_j$ e $\gamma \in H_i$ que têm um prefixo comum α tal que $\alpha \in \Omega_M(s_j) \cap \Omega_M(s_i)$ e $\lambda_M(s_j, \alpha) \neq \lambda_M(s_i, \alpha)$.

A operação $reset$ (representada como "r" nas sequências de entrada) é uma operação que "reinicia" corretamente a MEF, ou seja, leva a implementação ao seu estado inicial. A maior parte do métodos de geração utilizam essa operação para permitir que múltiplas sequências sejam aplicadas.

Dado um conjunto de sequências de entrada K, diz-se que dois estados $s_i, s_j \in S$ são K-equivalentes, denotado por $s_i \equiv_K s_j$, se para todo $\alpha \in K \cap \Omega_M(s_i) \cap \Omega_M(s_j)$, tem-se que $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$.

A implementação I está em conformidade com a especificação M se, e somente se para todo $\alpha \in \Omega_M(s_0) \cap \Omega_M(t_0)$, tem-se que $\lambda(s_0, \alpha) = \lambda(t_0, \alpha)$. Isso significa que, para cada sequência de entrada onde um comportamento de M seja definido, I comporta-se de maneira idêntica. Diz-se que a implementação é *quasi-equivalente* à especificação (Gill, 1962; Sidhu and Leung, 1989).

A.2 Propriedades de MEFs

Diversos métodos de geração requerem que as MEFs possuam determinadas propriedades para serem aplicados. As principais propriedades são apresentadas a seguir. Uma MEF é *completamente especificada* (ou *completa*) se existem transições definidas para todos os símbolos de entrada em cada estado da MEF. Caso contrário, a MEF é *parcialmente especificada* (ou *parcial*). Formalmente, uma MEF é completa se $D_A = S \times X$. Uma MEF é *fortemente conexa* se para cada par de estados (s_i e $s_j \in S$) existe uma sequência que leva a MEF M do estado s_i ao estado s_j. Uma MEF é dita ser *inicialmente conexa* se para cada estado $s \in S$ existe uma sequência que leva a MEF do estado inicial s_0 ao estado s. De uma forma geral, somente as MEFs inicialmente conectadas são consideradas nos estudos realizados, pois de acordo com Yannakakis and Lee (1995) qualquer estado inatingível a partir do estado inicial não afeta o comportamento da MEF. Uma MEF parcial é *reduzida* se seus estados, tomados par-a-par, são distinguíveis. Uma MEF completa é *minimal* se não possui par de estados equivalentes. Neste trabalho os termos *reduzida* e *minimal* são utilizados como sinônimos. Uma MEF é *determinística* se em cada estado, dada uma entrada, há somente uma única transição definida para um próximo estado caso contrário, a MEF é *não determinística*.

A.3 Domínio de Defeitos

Para um conjunto de sequências de teste gerado a partir de uma MEF, uma questão importante refere-se em como avaliar a *efetividade* (ou qualidade) do mesmo, ou seja, avaliar sua cobertura em relação aos defeitos revelados. Como, por um lado, infinitos defeitos são possíveis e, por outro, o conjunto de casos de teste deve ser finito, define-se um domínio de defeitos, representando o conjunto de possíveis defeitos que o teste deve revelar. Domínios de falha diferentes podem ser definidos para refletir características particulares de uma configuração de teste.

No teste baseado em MEFs, domínios de defeitos são definidos em função do número máximo de estados que a implementação pode ter. Assim, dado um número m, o domínio de defeitos é o conjunto de todas as MEFs com no máximo m estados. Note-se que em geral o domínio de defeitos, apesar de finito, possui um número muito grande de MEFs. Como a implementação é considerada uma caixa preta, esse número máximo não é conhecido, sendo que assume-se um valor que baseado em heurísticas. Trata-se de uma hipótese de teste (Chow, 1978; Petrenko and Yevtushenko, 2005; Hierons and Ural, 2006)

O conjunto de teste é m-completo se para qualquer implementação I do domínio de defeitos, I vai passar pelo teste, se e somente se, I estiver em conformidade com a especificação M.

Alguns métodos garantem a geração de conjuntos de teste m-completos, para qualquer $m \geq n$ pré-definido, sendo n o número de estados da especificação. Outros métodos apenas garantem para o caso de $m = n$, onde n é o número de estados da especificação. Além disso, alguns métodos não oferecem esse tipo de garantia.

A.4 Custo de Aplicação do Critério

O custo de aplicação de um método pode ser dividido em dois fatores principais. Por um lado, tem-se o custo para a geração das sequências de teste. Esse custo relaciona-se com a complexidade dos algoritmos utilizados durante o processo de geração. Os algoritmos de geração devem ser tratáveis, no sentido de que o

tempo necessário para gerar deve ser de ordem polinomial no tamanho da MEF. Por outro lado, tem-se o custo da execução das sequências de teste. Cada sequência de teste deve ser traduzida em entradas concretas para a implementação, a qual deve ser executada com essas entradas. Normalmente, o custo de execução é o principal fator na avaliação do custo da aplicação de um método, uma vez que é normalmente aceitável um método que demande mais tempo para geração das sequências de teste, se ele conseguir gerar um conjunto menor. Dessa forma, o custo de aplicação de um método é medido em termos do tamanho do conjunto de teste gerado, tanto no caso médio como no pior caso.

A forma usual de medir o custo é pela quantidade de símbolos de entrada presentes no conjunto, também conhecido como comprimento do conjunto. As sequências que são prefixos de outras sequências do conjunto não são contadas, pois ao se aplicar uma sequência, todos os seus prefixos já são necessariamente aplicados. Além disso, assume-se que para levar a implementação ao estado inicial, deve-se utilizar uma entrada adicional de *reset*. Assim, dado uma sequência de teste t, define-se o custo de t como sendo o comprimento de t mais 1. Dado um conjunto de sequências de teste T, define-se o comprimento como sendo a soma dos comprimentos de todas as sequências que não são prefixos de outras sequências em T.

A.5 Métodos de Geração

Nesta seção são apresentados os principais métodos de geração de casos de teste a partir de MEFs. O objetivo é fornecer uma visão geral dos métodos e da evolução histórica dos mesmos.

Embora os métodos possuam um objetivo comum (de verificar se uma implementação está correta com sua especificação), eles diferem com relação ao *custo* da geração das sequências de teste, *tamanho* do conjunto de teste e capacidade de detecção de defeitos (*eficácia*). Da mesma forma que as sequências geradas precisam detectar o máximo de defeitos existentes em uma implementação, elas devem ser relativamente pequenas para que seja possível sua aplicação na prática.

Nesta seção, são apresentados alguns métodos de geração de casos de teste, buscando ilustrar as diferenças apresentadas em relação ao conjunto de propriedades requeridas. Uma MEF, ilustrada na Figura A.1, será utilizada como exemplo para a geração dos casos de teste. Essa MEF possui quatro estados $\{S_1,\ S_2,\ S_3,\ S_4\}$, sendo S_1 o estado inicial, as entradas $X = \{x, y\}$ e as saídas $Y = \{0, 1\}$. A MEF também admite o conjunto de sequências $\{x, y, yy\}$ como o conjunto W, o conjunto state cover $Q = \{\epsilon, y, x, yy\}$, os identificadores de estado $W_1 = \{yy\}$, $W_2 = \{y\}$, $W_3 = \{x\}$ e $W_4 = \{x, yy\}$ e as famílias de separação $H_1 = \{x, yy\}, H_2 = \{x, y\}, H_3 = \{x\}$ e $H_4 = \{x, yy\}$.

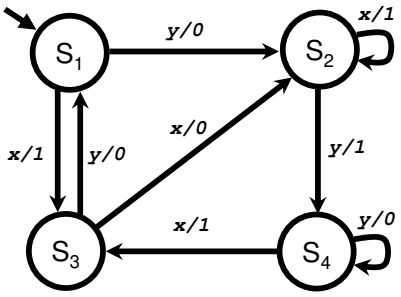


Figura A.1: Exemplo de MEF extraído de Dorofeeva et al. (2005b).

Cobertura de Estados e Transições Em Holzmann (1991) um algoritmo é proposto para o teste de conformidade.

O algoritmo de teste de conformidade funciona com a aplicação das mensagens $status$, $reset$ e set para todo estado $s \in S$, sendo $\alpha \in X$ da seguinte forma.

1. Aplique uma mensagem de $reset$ para trazer Mi ao seu estado inicial.
2. Aplique uma mensagem $set(s)$ para levar Mi ao estado s.
3. Aplique a entrada α.
4. Verifique se a saída produzida está em conformidade com a especificação M, ou seja, é igual $\lambda_S(s, \alpha)$.

5. Aplique a mensagem de $status$ e verifique se o estado final está em conformidade com a especificação Ms, ou seja, é igual $\delta_S(s, \alpha)$.

A *checking sequence* produzida pelo algoritmo é uma concatenação das sequências $reset$, $set(s)$, α e $status$ repetida para cada estado do conjunto de estados S e para cada símbolo de entrada do conjunto de símbolos de entrada X. Esse algoritmo é capaz de revelar qualquer defeito de saída e de transferência. No entanto, o algoritmo baseia-se na mensagem set, que por sua vez, pode não existir.

Para evitar o uso de mensagens set, uma sequência *transition tour* (TT) pode ser construída. Essa sequência percorre a máquina visitando cada estado e cada transição ao menos uma vez sem que ela precise ser reiniciada após a execução de cada teste. Pela aplicação do método TT, juntamente com uma mensagem de $status$ (inserida após cada entrada da sequência TT), uma *checking sequence* é obtida. Essa sequência de entrada consegue descobrir os defeitos de transferência e de saída. No entanto o método TT, proposto originalmente por Naito and Tsunoyama (1981), não utiliza a mensagem de $status$ e obtém somente uma *cobertura das transições*. Dessa forma, o método TT não garante a detecção de defeitos de transferência.

As mensagens de $status$ raramente estão disponíveis. Diversos métodos de geração de casos de teste utilizam algumas sequências de separação, ao invés da mensagem de $status$, para identificar os estados de uma MEF. Para a MEF da Figura A.1, o conjunto de casos de teste, gerado pelo método TT, é composto pelas sequências que realizam a cobertura das transições. O conjunto de teste obtido poderia ser $TS_{TT} = \{ryxyyxyxxyxy\}$ de tamanho 12.

Método DS O método DS, proposto por Gonenc (1970), baseia-se na sequência de distinção, ou seja, para a sua utilização é necessária que a MEF possua essa sequência. No entanto, segundo Gill (1962), tal sequência pode não existir mesmo para MEFs minimais.

É importante selecionar a menor sequência de distinção para que, consequentemente, se obtenha um conjunto menor de casos de teste. Seja X_d a sequência de distinção escolhida. O método resulta na geração de uma *checking sequence* pela composição de duas subsequências:

Sequências-α: Verificam todos os estados da MEF.

Sequências-β: Verificam todas as transições.

Primeiramente, o método consiste na geração das sequências-α. Para isso, um grafo (grafo-X_d) é construído de modo que cada estado da MEF seja representado por um nó. Para cada nó, existe uma aresta que o liga a um outro nó representando a aplicação de X_d. As sequências-α são geradas percorrendo-se o grafo sem repetir as arestas.

Em seguida, as sequências-β são produzidas de forma semelhante às sequências-α. Um outro grafo é produzido (grafo-β), no entanto, as arestas representam sequências da forma $x_i.X_d$. As sequências-β são geradas obtendo-se uma cobertura das arestas do grafo-β.

Considerando a MEF da Figura A.1 com a sequência de distinção $X_d = yyy$, o método DS é ilustrado a seguir.

Para a geração das sequências-α o grafo-X_d, ilustrado na Figura A.2, é construído. Para cada nó, as transições referentes à aplicação de X_d são representadas.

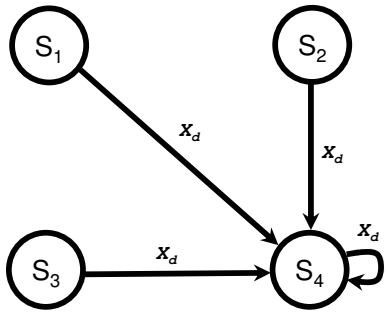


Figura A.2: Grafo-X_d.

No início, um estado que não é destino de nenhuma aresta (estado origem) é escolhido arbitrariamente. Por exemplo, o estado S_1 é escolhido e marcado como "reconhecido". Aplica-se a sequência X_d atingindo o estado S_4 que também é marcado como "reconhecido". Aplica-se X_d atingindo o estado S_4 novamente. Assim, um novo estado origem deve ser selecionado, mas antes disso, aplica-se novamente X_d para verificar se o estado atingido foi realmente o estado

S_4. A partir de S_4 aplica-se x que leva a MEF ao novo estado origem S_3. Estando no estado S_3, repete-se o procedimento anterior. A sequência-α obtida é: $\{yyy\ yyy\ yyy\ x\ yyy\ yyy\ xx\ yyy\ yyy\}$.

Em seguida, para a construção das sequências-β o grafo-β (Figura A.3(a)) é criado. Considerando que as sequências-α já foram aplicadas e, dessa forma, todos os estados já foram verificados, duas reduções podem ser realizadas no grafo-β. A primeira refere-se à última transição da aplicação de X_d. Por exemplo, aplicando-se X_d ao estado S_1 a MEF passa pelos estados S_2, S_4 e S_4. O último passo pode ser descartado, pois essa verificação já foi realizada na construção das sequências-α. Desse modo, a transição de S_4 com a entrada y pode ser retirada do grafo-β. De maneira semelhante, a transição de S_2 com a entrada y pode ser retirada do grafo-β.

A segunda redução refere-se à última transição da sequência incluída para ligar os estados de origem. Por exemplo, a sequência x ligou o estado S_4 ao estado S_3, então a transição de S_4 com a entrada x pode ser retirada do grafo-β. Do mesmo modo, a sequência xx ligou o estado S_4 ao estado S_2, passando pelo estado S_3. Dessa forma, a transição de S_3 com a entrada x também pode ser retirada do grafo-β. O grafo-β reduzido é ilustrado na Figura A.3(b).

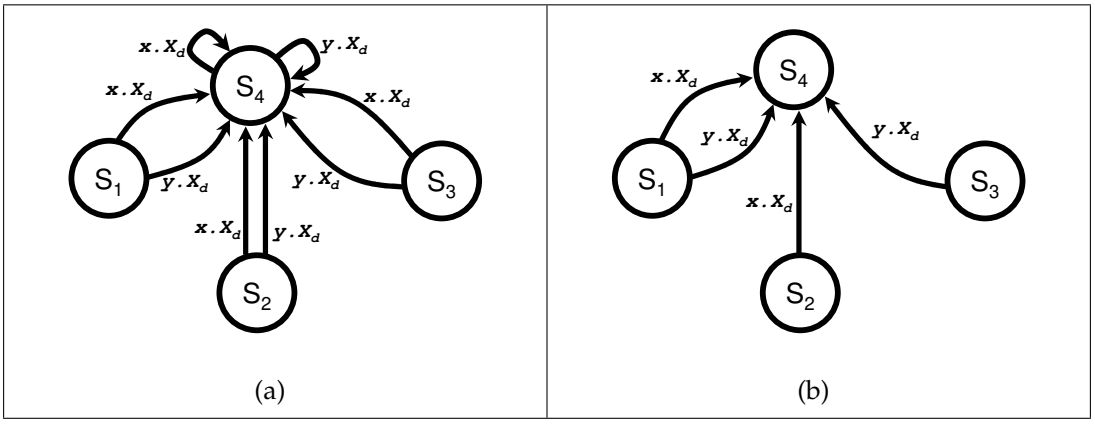


Figura A.3: Grafo-β e Grafo-β reduzido.

Percorre-se o grafo-β reduzido para a obtenção da sequência-β. A sequência-β obtida é: $\{xyyy\ xy\ yyyy\ xx\ xyyy\ x\ yyyy\}$.

O conjunto de casos de teste resultante da aplicação do método DS é: $TS_{DS} = \{yyy\ yyy\ yyy\ x\ yyy\ yyy\ xx\ yyy\ yyy\ xyyy\ xy\ yyyy\ xx\ xyyy\ x\ yyyy\}$ de tamanho 45.

É importante salientar que trabalhos vêm sendo desenvolvidos em relação à redução de *checking sequences*. No trabalho de Ural et al. (1997), um método para a construção de *checking sequences* é proposto e, da mesma forma que o método DS, é aplicável somente para MEFs que possuam uma sequência de distinção. No trabalho de Hierons and Ural (2002) uma melhoria é proposta ao método criado em Ural et al. (1997) para a construção de *checking sequences* de tamanho mínimo.

A partir dessa melhoria, houve uma redução no tamanho das *checking sequences* geradas a partir de MEFs determinísticas, minimais e completamente especificadas. A *checking sequence* é produzida com base nos conjuntos A (conjunto de sequências) e E_c (conjunto de transições). No trabalho de Hierons and Ural (2006) investiga-se a escolha desses conjuntos. Os autores demonstram como o conjunto A deve ser escolhido para minimizar a soma dos tamanhos das sequências e como essa etapa deve ser adaptada para a geração de um conjunto E_c ótimo. Os resultados obtidos apontam uma redução de 25 a 40% das *checking sequences*.

Método W Um dos métodos mais conhecidos para a geração de sequências de teste é o método W (*Automata Theoretic*) proposto por Chow, em 1978. O método W não é aplicado a MEFs parciais, considerando apenas MEFs inicialmente conectadas, completamente especificadas, minimais e determinísticas. Esse método consiste em gerar dois conjuntos de sequências e concatená-los de forma a obter sequências de entrada para o teste de determinada MEF. Esses dois conjuntos são:

P**:** Conjunto de sequências que percorre cada transição ao menos uma vez.

T**:** Conjunto de sequências capaz de identificar qual é o estado da máquina.

O conjunto T é gerado a partir de um conjunto de caracterização (conjunto W). Em seguida, é estimado o número m de estados da máquina a ser testada. Se o número estimado for igual ao número de estados n da máquina real, então $T = W$, senão tem-se $T = \bigcup_{i=0}^{m-n} (X^i \bullet W)$, onde X^i é o conjunto de todas as sequências com i entradas, e $A \bullet B = \{\alpha\beta \mid \alpha \in A \wedge \beta \in B\}$. Ao fim, a sequência de teste gerada dá-se pela concatenação de P com T.

As sequências desse conjunto são executadas uma a uma na máquina, gerando as saídas que são analisadas posteriormente.

Em suma, o método W consiste em três passos principais:

1. Estima-se um número máximo (m) de estados que a implementação possa conter.
2. Geração das sequências de teste que garantem que cada transição foi implementada corretamente.
3. Verificação das respostas geradas pelas sequências de teste produzidas na segunda etapa.

Se a implementação da MEF (a máquina em teste) gerar saídas corretas a partir das sequências de entrada geradas pelo método W, esta máquina está correta, pois o método é confiável para testar estruturas de controle modeladas por uma MEF (Chow, 1978). Contudo, o método W produz muitas sequências de entrada para serem testadas, o que pode promover um alto custo para a realização da etapa de teste.

A aplicação do método W na MEF da Figura A.1 é ilustrada a seguir. Considerando $m = n$ tem-se o conjunto $T = W = \{x, y, yy\}$. Considera-se o conjunto *transition cover* $P = \{\epsilon,\ x,\ y,\ xx,\ xy,\ yy,\ yx,\ yyy,\ yyx\}$. Pela concatenação de P com T obtém-se as sequências $\{x,\ y,\ yy,\ xx,\ xy,\ xyy,\ yx,\ yy,\ yyy,\ xxx,\ xxy,\ xxyy,\ xyx,\ xyy,\ xyyy,\ yyx,\ yyy,\ yyyy,\ yxx,\ yxy,\ yxyy,\ yyyx,\ yyyy,\ yyyyy,\ yyxx,\ yyxy,\ yyxyy\}$.

Com a retirada das sequências que são prefixos de outras, a aplicação do método W na MEF da Figura A.1 resulta no conjunto $TS_W = \{rxxx,\ rxxyy,\ rxyx,\ rxyyy,\ ryxx,\ ryxyy,\ ryyxx,\ ryyxyy,\ ryyyx,\ ryyyyy\}$ de tamanho 49.

Método Wp Fujiwara et al. (1991) propuseram o método Wp (partial W) que é um aprimoramento do método W. A principal vantagem do método Wp em relação ao W é que ele utiliza um subconjunto do conjunto W para a criação das sequências de teste, e, assim, obtém-se uma quantidade reduzida de casos de teste para serem utilizados.

O método Wp, semelhante ao W, também opera em MEFs completas. Esse método possui o mesmo poder do método W na detecção de defeitos, mas produz

um menor conjunto de sequências de entradas (Fujiwara et al., 1991). O método basicamente consiste em duas fases:

Fase 1: É verificado se todos os estados definidos na especificação também são encontrados na implementação.

Fase 2: Todas as transições definidas na especificação e que não foram testadas na fase 1 são verificadas.

Um conjunto *transition cover* P que cobre todas as transições da MEF é determinado e identifica-se um subconjunto *state cover* Q que cobre todos os estados da MEF. Para cada estado $s_i \in S$ da especificação determina-se um conjunto de identificação W_i, que distingue o estado s_i de todos os demais. A união de todos os conjuntos W_i resulta no conjunto W e diferentes casos de testes podem ser gerados dependendo da escolha dos conjuntos P, Q e W_i.

Na primeira fase, os casos de teste resultam da concatenação dos conjuntos Q e W. Se o teste obtiver sucesso significa que o número de estados da implementação é igual ao número de estados da especificação.

Na segunda fase, os casos de teste são gerados a partir da concatenação das sequências do conjunto P, menos as sequências do conjunto Q, com o conjunto W_i correspondente ao estado atingido após a execução de cada sequência, ou seja, $R = P - Q$ e $R \otimes W = \bigcup_{p \in R} \{p\} \bullet W_i$. A operação $R \otimes W$ resulta em um conjunto formado pela união da concatenação das sequências do conjunto R com o conjunto de identificação W_i. Dessa forma, obtém-se um conjunto de casos de teste menor em relação ao conjunto gerado pelo método W, pois a concatenação ocorre com um subconjunto W_i ao invés de ocorrer com o conjunto W.

Para a MEF da Figura A.1, a aplicação do método Wp é ilustrado a seguir. Na primeira fase, considerando o conjunto *state cover* Q, as sequências são geradas pela concatenação de Q com W. Dessa forma, como resultado da primeira fase tem-se as sequências $\{x,\ y,\ yy,\ yx,\ yy,\ yyy,\ xx,\ xy,\ xyy,\ yyx,\ yyy,\ yyyy\}$.

Na segunda fase, considerando o conjunto *transition cover* $P = \{\epsilon,\ x,\ y,\ xx,\ xy,\ yy,\ yx,\ yyy,\ yyx\}$, as sequências são geradas pela concatenação do conjunto P, menos o conjunto Q, com o conjunto W_i de cada estado S_i atingido. Tem-se $R = P - Q = \{xx,\ xy,\ yyy,\ yyx,\ yx\}$. Realizando a operação $R \otimes W$

obtém-se as sequências da forma: $\{xx.W_2,\ xy.W_1,\ yyy.W_4,\ yyx.W_3,\ yx.W_2\}$. Realizando as substituições necessárias, as sequências obtidas são: $\{xxy,\ xyyy,\ yyyx,\ yyyyy,\ yyxx,\ yxy\}$.

Com a retirada das sequências que são prefixos de outras, a aplicação do método Wp na MEF da Figura A.1 produz o conjunto $TS_{Wp} = \{rxxy,\ rxyyy,\ ryxy,\ ryyxx,\ ryyyx,\ ryyyyy\}$ de tamanho 29.

Método *State Counting* O método *State Counting* (SC), proposto por Petrenko and Yevtushenko (2005), atinge os mesmos objetivos do método W em relação à efetividade, ou seja, o método garante a cobertura completa de defeitos existentes na implementação de uma MEF parcial.

De um modo geral, o método *State Counting* utiliza um algoritmo que expande as sequências de teste a partir de um estado da MEF até que seja atingida uma condição que permita verificar que todos os defeitos já foram identificados. Por exemplo, se um estado é visitado mais do que m vezes, sendo m o número de estados da MEF, pode-se parar de expandir a sequência, uma vez que, desse ponto em diante, o comportamento começará a se repetir. Os autores provam que, utilizando-se as relações de quasi-equivalência entre estados e sequências capazes de distinguir pares de estado, pode-se determinar a parada da expansão da sequência sem que seja necessário atingir o limite de m visitas a um estado.

O método *State Counting* pode ser utilizado com MEFs parciais e gera um conjunto de casos de teste que pode ser usado para identificar todos os possíveis defeitos, sendo, dessa forma, mais eficiente que o método HSI e mais amplamente aplicável que os métodos W e Wp. Contudo, verifica-se que, em geral, o número de casos de teste gerados é elevado. Esse método gera um conjunto de casos de teste completo a partir de MEF parciais e não reduzidas.

Método HSI O método HSI (Petrenko et al., 1993), semelhante ao método Wp, também é uma modificação do método W. Ele garante a cobertura completa de defeitos existentes sendo aplicável em qualquer especificação reduzida, seja ela completa ou parcial. Esse método utiliza o conceito de família de separação. Uma

família de separação é um conjunto de identificadores de estado $H_j, s_j \in S$, que satisfaz a seguinte condição:

Com o objetivo de testar a conformidade de uma implementação I em relação à especificação M, o método consiste basicamente em, dado um *transition cover* P, anexar a cada sequência $\alpha \in P$, o conjunto de separação H_j, tal que $s_j = \delta(s_0, \alpha)$.

Se a MEF resultar em respostas corretas para as sequências produzidas em ambas as fases, pode-se considerar que ela está em conformidade com sua especificação.

Para a MEF da Figura A.1, a aplicação do método HIS é ilustrado a seguir, considerando o conjunto *state cover* Q e as famílias de separação H_1, H_2, H_3, H_4.

Na primeira fase (Identificação de Estados) as sequências geradas são da forma: $\{\epsilon.H_1,\ \ y.H_2,\ \ x.H_3, yy.H_4\}$ resultando nas sequências $\{x,\ yy,\ yx,\ yy,\ xx,\ yyx,\ yyyy\}$.

Na segunda fase (Teste de Transições) as sequências geradas são da forma: $\epsilon.x.H_3,\ \ \epsilon.y.H_2,\ \ y.x.H_2,\ \ y.y.H_4,\ \ x.x.H_2,\ \ x.y.H_1,\ \ yy.x.H_3,\ \ yy.y.H_4$. Realizando as substituições necessárias, essa fase gera as sequências: $\{xx,\ xy,\ yx,\ yy,\ yxx,\ yxy,\ yyx,\ yyyy,\ xxx,\ xxy,\ xyx,\ xyyy,\ yyxx,\ yyyx,\ yyyyy\}$.

Com a retirada das sequências que são prefixos de outras, o método HIS gera o conjunto $TS_{HIS} = \{rxxx,\ rxxy,\ rxyx,\ rxyyy,\ ryxx,\ ryxy,\ ryyxx,\ ryyyx,\ ryyyyy\}$ de tamanho 41.

Método H O método H (Dorofeeva et al., 2005a) é uma melhoria do método HIS. A idéia é não utilizar, *a priori*, os identificadores de estados gerados. Os identificadores de estado são construídos com base nos casos de teste já derivados com o intuito de distinguir-se os estados finais das transições. No trabalho de Dorofeeva et al. (2005a) o método H, proposto originalmente para MEFs completas e determinísticas, é estendido para MEFs determinísticas parciais.

Os autores também estenderam o método H para máquinas parciais não determinísticas. Segundo os autores, o método H, bem como o HIS, gera um conjunto de teste completo, sendo aplicável em qualquer especificação reduzida completa ou parcial. No entanto, o tamanho do conjunto dos casos de teste gerado depende da ordem na qual as transições são verificadas. Os autores afir-

mam que um procedimento para determinar a ordem de escolha das transições está sendo incorporado ao método para a obtenção de sequências menores.

Considerando a MEF da Figura A.1, para identificar os estados, o método H utiliza os identificadores de estado $W_1,\ W_2,\ W_3, W_4$ gerando as sequências $\{yx,\ xx,\ yyx,\ yyyy\}$. Para verificar as transições, os identificadores de estados são gerados. Por exemplo, seja a transição do estado S_3 para o estado S_2 com a entrada x. Ao invés de se utilizar a sequência $x.x.H_2$ como no método HIS, utiliza-se a sequência $x.x.y$.

Para a MEF da Figura A.1, a aplicação do método H produz o conjunto $TS_H = \{rxxy,\ rxyyy, ryxy,\ ryyxx,\ ryyyyyy\}$ de tamanho 25.

Método SPY O método SPY (Simão et al., 2009c), baseado em condições de suficiência propostas em (Simão and Petrenko, 2010b), reduz o tamanho dos conjuntos de teste pela distribuição dos identificadores entre várias sequências. A completude do conjunto é garantida pela verificação de que as várias sequências levam a um mesmo estado na implementação. Apesar de a implementação ser uma caixa preta, a suposição de ela comporta-se como uma MEF com um número máximo, conhecido de estados e de que ela é deterministica permite concluir que algumas sequências devem levar ao mesmo estado. Assim, o método SPY evita que muitas sequências seja adicionadas ao conjunto, aumentando seu tamanho. Note-se que o método não especifica quais identificadores devem ser utilizados. Dessa forma, ele pode ser combinado com os métodos Wp, HSI ou H.

A.5.1 Comparação entre os Métodos de Geração

Para que o testador escolha um método de geração com o objetivo de aplicá-lo em alguma especificação baseada em MEF, é necessário que algumas características sejam observadas. Essas características referem-se à exigência de cada método para que a MEF possua certas propriedades, ao tamanho das sequências geradas e à aplicabilidade de cada um.

Na Tabela A.1 é fornecida uma comparação entre os métodos apresentados nesta seção. Todos os métodos são aplicados às MEFs determinísticas, fortemente

conexas, completas e minimais. Dessa forma, na Tabela A.1 são apresentadas outras características das MEFs em que alguns métodos ainda podem ser aplicados.

Tabela A.1: Comparação entre os métodos de geração.

	TT	DS	W	Wp	SC	HSI	H	SPY
Não-minimal	✓				✓			
Parcial	✓				✓	✓	✓	✓
Não-determinística							✓	
Cobertura Completa		✓	✓	✓	✓	✓	✓	✓
Tamanho do Conjunto	12	45	49	29	39	39	25	25

Dentre os métodos apresentados, o método *State Counting* é o único que pode ser aplicado às MEFs não reduzidas e que obtém um conjunto de casos de teste completo. De acordo com Petrenko and Yevtushenko (2005), tem-se trabalhado para realizar a generalização desse método para que seja aplicado às MEFs não determinísticas.

Para a aplicação do método W, o conjunto de caracterização (conjunto W) deve existir, sendo que ele sempre existe em MEFs minimais. O método DS fica restrito à existência da sequência de distinção. Os métodos H e SPY, são os mais recentes, incorporando estratégias possibilitadas por novas condições de suficiência identificados (por exemplo, (Dorofeeva et al., 2005a; Simão et al., 2009c).

Apêndice B

Teste de Programas Paralelos: Definições e Exemplos

A seguir são apresentados os conceitos do teste de programas paralelos. Primeiramente, apresenta-se o modelo *Parallel Control Flow Graph* (PCFG) (Souza et al., 2008) e em seguida, os critérios definidos são ilustrados. Este apêndice foi baseado na Seção 5.2 da dissertação de mestrado de Endo (2008).

O modelo PCFG foi definido para capturar o fluxo de controle, dados e comunicação em programas paralelos baseados em passagem de mensagens. O modelo considera um número n fixo e conhecido de processos dado pelo conjunto $Prog = \{p^0, p^1, \ldots p^{n-1}\}$. A comunicação entre esses processos é feita por meio das primitivas *send* e *receive*. Cada processo p possui seu próprio grafo de fluxo de controle CFG^p (Rapps and Weyuker, 1985).

O PCFG é composto pelos GFCs dos processos e pela representação da comunicação entre os processos. N e E representam os conjuntos de nós e arcos, respectivamente. Cada nó n_i no processo p é representado com a notação n_i^p. Dois subconjuntos de N são definidos: N_s e N_r, compostos de nós que são associados às primitivas *send* e *receive*, respectivamente. O conjunto E também possui dois subconjuntos: E_i^p contém os arcos intra-processo (internos) do processo p e E_s contém os arcos inter-processos (representam a comunicação) do $PCFG$. A criação dos arcos inter-processos pode ser realizada utilizando uma abordagem

conservativa, em que são combinados todos os nós *send* com todos os nós *receive*, exceto os que estão no mesmo processo. Um problema desse tipo de abordagem é o grande número de arcos inter-processos que são gerados.

Um caminho π^p em um CFG^p é chamado caminho intra-processo. Um caminho inter-processos possui pelo menos um arco inter-processos e é dado por um conjunto de caminhos, $\Pi = (\pi^0, \pi^1, \ldots \pi^k, S)$, onde S é o conjunto de arcos inter-processos (pares de sincronização) que foram executados.

Uma variável é geralmente definida em atribuições e comandos de entrada. No contexto de ambientes de passagem de mensagens, uma variável também pode ser definida em funções de comunicação como o *receive*. Essas funções definem uma ou mais variáveis com valores recebidos na mensagem (Souza et al., 2008). Um conjunto de variáveis que são definidas no nó n_i^p é representado por $def(n_i^p)$, ou seja, $def(n_i^p) = \{x \mid x \text{ é uma variável definida em } n_i^p\}$. Um caminho $\pi = (n_1, n_2, ..., n_{k-1}, n_k)$ é livre de definição com respeito à variável x do nó n_1 para o nó n_k ou arco (n_{k-1}, n_k), se $x \in def(n_1)$ e $x \notin def(n_i)$, para $i = 2..k-1$.

Além dos tradicionais uso predicativo (*p-uso*) e uso computacional (*c-uso*) de variáveis, o modelo PCFG adiciona o **uso de comunicação** (*s-uso*). Um *s-uso* ocorre quando uma variável é usada em uma sentença de comunicação, relacionada a um arco inter-processos. Essas associações são definidas a seguir:

- Um **c-uso** é definido pela tripla $(n_i^p, n_j^p, x) \mid x \in def(n_i^p)$ e, n_j^p possui um *c-uso* de x e, existe um caminho livre de definição em relação à x de n_i^p para n_j^p.

- Um **p-uso** é definido pela tripla $(n_i^p, (n_j^p, n_k^p), x) \mid x \in def(n_i^p)$ e, (n_j^p, n_k^p) possui um *p-uso* de x e, existe um caminho livre de definição em relação à x de n_i^p para (n_j^p, n_k^p).

- Um **s-uso** é definido pela tripla $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), x) \mid x \in def(n_i^{p1})$ e, (n_j^{p1}, n_k^{p2}) possui um *s-uso* de x e, existe um caminho livre de definição em relação à x de n_i^{p1} para (n_j^{p1}, n_k^{p2}).

- Um **s-c-uso** é definido por $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), n_l^{p2}, x^{p1}, x^{p2})$, onde existe uma associação *s-uso* $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), x^{p1})$ e uma associação *c-uso* $(n_k^{p2}, n_l^{p2}, x^{p2})$.

- Um **s-p-uso** é definido por $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), (n_l^{p2}, n_m^{p2}), x^{p1}, x^{p2})$, onde existe uma associação *s-uso* $(n_i^{p1}, (n_j^{p1}, n_k^{p2}), x^{p1})$ e uma associação *p-uso* $(n_k^{p2}, (n_l^{p2}, n_m^{p2}), x^{p2})$.

Para ilustrar o modelo, a seguir é apresentado o exemplo GCD. Esse exemplo é implementado usando a biblioteca PVM (Listagem B.1 e Listagem B.2). O programa utiliza quatro processos (p^m, p^0, p^1, p^2) para calcular o máximo divisor comum entre três números. O processo mestre p^m cria os processos escravos $p^0, p^1\ and\ p^2$, que executam o código "gcd.c". Cada escravo espera o recebimento de dois valores enviados pelo processo p^m e calculam o máximo divisor comum para esses valores. Ao final, os processos escravos enviam o valor calculado para o processo mestre.

Listing B.1: Programa GCD em PVM - processo mestre.

```
/* 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();      }
```

Listing B.2: Programa GCD em PVM - processo escravo.

```
/* 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();}
```

O $PCFG$ é apresentado na Figura B.1. O número à esquerda do código-fonte (Listagem B.1 e B.2) representa o nó no grafo associado a cada comando. Arcos inter-processos são representados por arcos tracejados.

B.1 Critérios de Teste

Durante a atividade de teste, é essencial avaliar a qualidade dos testes realizados. Um critério de teste define propriedades ou requisitos que precisam ser testados

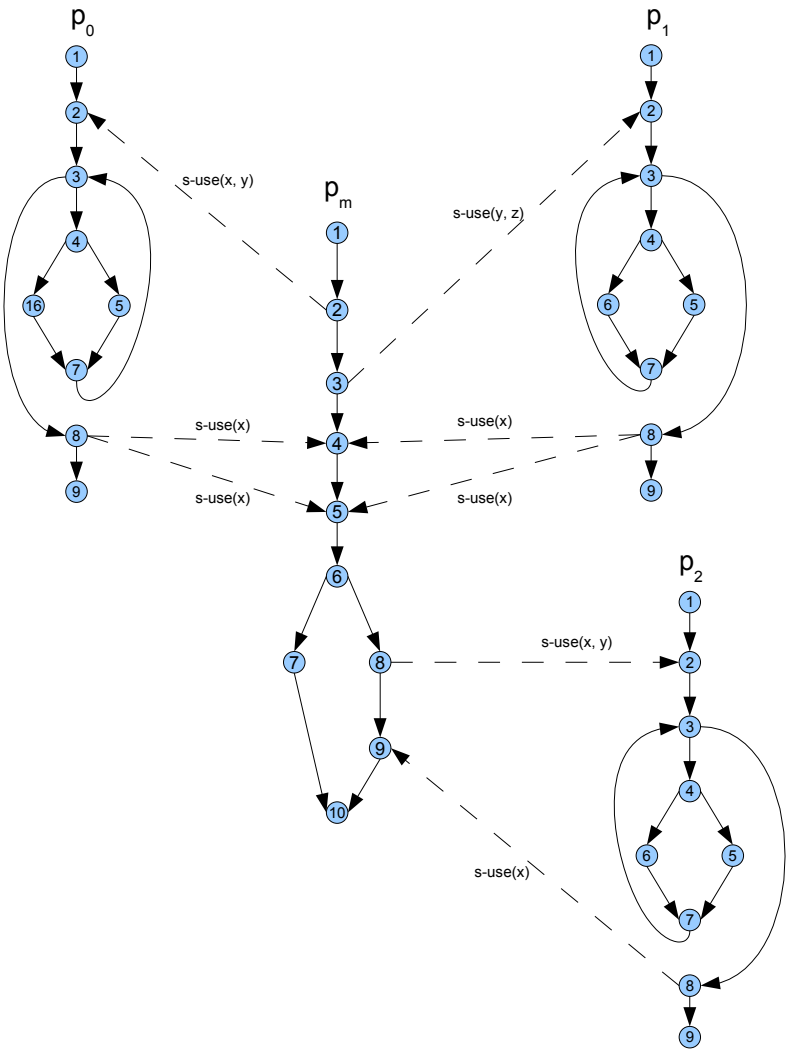


Figura B.1: PCFG para o programa GCD (Souza et al., 2008).

para garantir a qualidade do software (Rapps and Weyuker, 1985). Critérios de teste são usados para avaliar sistematicamente casos de teste e guiar a seleção de casos de teste. Baseado no modelo PCFG, Souza et al. (2008) definiram um conjunto de critérios de cobertura, listados a seguir.

- **Todos-Nós-s**: os casos de teste devem exercitar cada nó $n_i^p \in N_s$.
- **Todos-Nós-r**: os casos de teste devem exercitar cada nó $n_i^p \in N_r$.
- **Todos-Nós**: os casos de teste devem executar todas as atividades em todos os processos.
- **Todos-Arcos-s**: os casos de teste devem executar no mínimo uma vez cada comunicação entre os processos.
- **Todos-Arcos**: os casos de teste devem executar todos os desvios de execução e comunicação entre processos.
- **Todos-s-usos**: os casos de teste devem executar todas as associações *s-uso*.
- **Todos-s-c-usos**: os casos de teste devem executar todas as associações *s-c-uso*.
- **todos-s-p-usos**: os casos de teste devem executar todas as associações *s-p-uso*.

Durante a análise de cobertura, que consiste basicamente em determinar o percentual de elementos requeridos de critério de teste que foram exercitados pelo conjunto de casos de teste, é fundamental o conhecimento sobre as limitações inerentes à atividade de teste (Maldonado et al., 2004). Sabe-se que alguns elementos requeridos podem ser não executáveis, e em geral, determinar a não executabilidade de um dado requisito de teste é feita de forma manual.

Apêndice C

A. S. Simão, A. Petrenko. Checking Completeness of Tests for Finite State Machines. IEEE Transactions on Computers, v. 59, p. 1023-1032, 2010

Checking Completeness of Tests for Finite State Machines

Adenilso Simao and Alexandre Petrenko

Abstract—In testing from a Finite State Machine (FSM), the generation of test suites which guarantee full fault detection, known as complete test suites, has been a long-standing research topic. In this paper, we present conditions that are sufficient for a test suite to be complete. We demonstrate that the existing conditions are special cases of the proposed ones. An algorithm that checks whether a given test suite is complete is given. The experimental results show that the algorithm can be used for relatively large FSMs and test suites.

Index Terms—Finite State Machine, test analysis, fault coverage, test completeness conditions, test generation.

1 Introduction

Test generation from a Finite State Machine (FSM) is a long-standing research problem, with numerous contributions over decades. Since the seminal work of Moore [12] and Hennie [8], several methods have been proposed to generate a test suite with full fault detection capability, i.e., a test suite which provides full coverage of the set of all possible FSMs with a certain number of states that model implementations of a given specification FSM; such test suites have complete fault coverage and, in this sense, are complete [1], [2], [4], [5], [9], [10], [15], [17], [18], [20]. These methods rely on sufficient conditions for test suite completeness. The conditions appear either explicitly in the methods or implicitly in the proof of their correctness.

The generation methods usually require the existence of sequences which identify states in the specification FSM based on their outputs. If the FSM is completely specified and has a diagnostic sequence, a complete test suite with a single sequence can be generated, as in, e.g., [5], [9], [10], [8], [18]. The sufficient conditions underlying the correctness proof of these methods are captured in a theorem presented in [18]. However, a diagnostic sequence may not exist for an arbitrary reduced FSM. In this case, methods which do not require the existence of a diagnostic sequence can be used, such as those presented in [17], [20]. These methods are applicable to any reduced FSMs and generate test suites with multiple sequences, as they rely on the availability of a reliable reset operation. The related sufficient conditions are summarized in [14] and refined in [2].

Besides supporting the definition of generation methods, sufficient conditions for test completeness can be used to address other related issues, namely, the analysis of the fault coverage of a test suite and test minimization. Completeness of a test suite can be established by exhaustive approaches which explicitly enumerate either all possible faulty FSMs, as in, e.g., [16] or all minimal forms of the partially specified FSM representing a test suite as a tree (see [19], [6]). By their nature, these approaches do not scale well. This fact explains why approaches which reduce the task of deciding whether a given test suite has complete fault detection capability to checking the satisfaction of sufficient conditions appear to be more practical even if they cannot give a definitive answer when the conditions are not satisfied.

The relevance of investigating sufficient completeness conditions is thus twofold. On one hand, weakening sufficient conditions can allow for improvement in methods for test generation, obtaining shorter tests of a proven fault detection capability. On the other hand, weaker sufficient conditions can be used to prove completeness of a much larger class of tests, as well as to further minimize existing complete tests.

In this paper, we present sufficient conditions for test suite completeness that are weaker than the ones known in the literature. We consider the case when implementation FSMs have at most as many states (n) as the specification FSM. Test completeness in this case is usually called n-completeness. We introduce the notion of *confirmed* sequence set. A set of input sequences is confirmed with respect to a test suite T and an FSM M if sequences leading to a same state in M also lead to a same state in any FSM that has the same output responses to T and has as many states as M. We show that if there exists a confirmed set which includes the empty sequence and traverses each defined transition, then a test suite is n-complete. We also demonstrate that the proposed conditions generalize both those proposed in [18] (which do not need a reliable reset but require a diagnostic sequence) and in [2] (which need a reliable reset but do not require a diagnostic sequence). We also present an approach for determining confirmed sets and elaborate an algorithm for analyzing test completeness. The effectiveness of the algorithm is demonstrated by experimenting with randomly generated FSMs with up to 500 states and test suites with up to 300,000 inputs.

- *A. Simao is with the Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, Av. Trabalhador São-carlense, 400—Centro, PO Box 668, São Carlos 13560-970, SP, Brazil. E-mail: adenilso@icmc.usp.br.*
- *A. Petrenko is with the Centre de Recherche Informatique de Montreal (CRIM), 405, Avenue Ogilvy, Bureau 101, Montreal, Quebec H3N 1M3, Canada. E-mail: petrenko@crim.ca.*

Manuscript received 24 Sept. 2007; revised 26 May 2008; accepted 26 Oct. 2009; published online 14 Jan. 2010.
Recommended for acceptance by S. Tragoudas.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-09-0480.
Digital Object Identifier no. 10.1109/TC.2010.17.

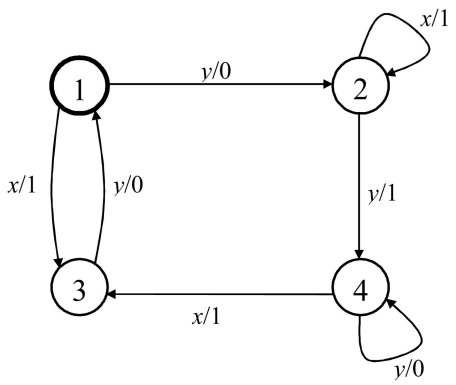


Fig. 1. A partial FSM with the initial state 1.

This paper is organized as follows: In Section 2, we provide the necessary basic definitions. In Section 3, we define the notion of confirmed sets, state sufficient conditions for a test suite to be n-complete, based on the existence of confirmed sets and elaborate an approach for determining confirmed sets. An algorithm for checking n-completeness is presented in Section 4. We then demonstrate in Section 5 that all known sufficient conditions for n-completeness are special cases of the conditions proposed in this paper. The results of the experimental evaluation of the formulated conditions and method to check them are discussed in Section 6. Section 7 concludes the paper.

2 Definitions

A Finite State Machine is a deterministic Mealy machine, which can be defined as follows:

Definition 1. *A* Finite State Machine *(FSM) M is a 7-tuple $(S,\ s_0,\ I,\ O,\ D,\ \delta,\ \lambda)$, where*

- *S is a finite set of states with the initial state s_0,*
- *I is a finite set of inputs,*
- *O is a finite set of outputs,*
- *$D \subseteq S \times I$ is a specification domain,*
- *$\delta : D \to S$ is a transition function, and*
- *$\lambda : D \to O$ is an output function.*

If $D = S \times I$, then M is a *complete* FSM; otherwise, it is a *partial* FSM. As M is deterministic, a tuple $(s, x) \in D$ determines uniquely a (*defined*) *transition* of M in state s. For simplicity, we use (s, x) to denote the transition, thus omitting its output and final state. A string $\alpha = x_1 \ldots x_k,\ \alpha \in I^*$, is said to be a *defined* input sequence for state $s \in S$, if there exist $s_1, \ldots, s_{k+1}$, where $s_1 = s$, such that $(s_i, x_i) \in D$ and $\delta(s_i, x_i) = s_{i+1}$, for all $1 \leq i \leq k$. We use $\Omega(s)$ to denote the set of all defined input sequences for state s and Ω_M as a shorthand for $\Omega(s_0)$, i.e., for the input sequences defined for the initial state of M and, hence, for M itself. Fig. 1 shows the example of a partial FSM.

We extend the transition and output functions from input symbols to defined input sequences, including the empty sequence ε, as usual: for $s \in S, \delta(s, \varepsilon) = s$ and $\lambda(s, \varepsilon) = \varepsilon$; and for input sequence α and input x, $\delta(s, \alpha x) = \delta(\delta(s, \alpha), x)$ and $\lambda(s, \alpha x) = \lambda(s, \alpha)\lambda(\delta(s, \alpha), x)$. Moreover, we extend the transition function to sets of defined input sequences. Given an FSM M, a state s of M, and a set of defined input sequences $C \subseteq \Omega(s)$, we define $\delta(s,\ C)$ to be the set of states reached by the sequences in C, i.e., $\delta(s,\ C) = \{\delta(s, \alpha) \mid \alpha \in C\}$. For simplicity, we slightly abuse the notation and write $\delta(s, C) = s'$, whenever $\delta(s, C) = \{s'\}$. Given sequences $\alpha, \beta, \gamma \in I^*$, if $\beta = \alpha\gamma$, then α is a *prefix* of β; if, moreover, γ is not empty, then α is a *proper* prefix of β. A set of sequences A is *prefix-closed*, if for each sequence $\beta \in A$, it holds that A contains all prefixes of β.

An FSM M is said to be *initially connected*, if for each state $s \in S$, there exists a defined input sequence $\alpha \in \Omega_M$, called a *transfer* sequence for state s, such that $\delta(s_0, \alpha) = s$. In this paper, only initially connected machines are considered, since any state that is not reachable from the initial state can be removed without changing the machine's behavior.

Two states $s,\ s' \in S$ are *distinguishable*, if there exists $\gamma \in \Omega(s) \cap \Omega(s')$, such that $\lambda(s, \gamma) \neq \lambda(s', \gamma)$. We say that γ *distinguishes* s and s'. If a sequence γ distinguishes each pair of distinct states, then γ is a *diagnostic* sequence. Given a set $C \subseteq \Omega(s) \cap \Omega(s')$, states s and s' are *C-equivalent*, if $\lambda(s, \gamma) = \lambda(s', \gamma)$, for all $\gamma \in C$. We finally define distinguishability and C-equivalence of machines as a corresponding relation between their initial states. An FSM M is said to be *reduced*, if all states are pairwise distinguishable.

3 Complete Test Suite and Sufficient Conditions

We consider only deterministic machines in this paper. Thus, a test case can be defined using just inputs, as expected outputs are uniquely determined from the inputs by a given specification FSM.

Definition 2. *A defined input sequence of FSM M is called a* test case *(or simply a* test*) of M. A* test suite *T of M is a finite prefix-closed set of tests of M. A test $\alpha \in T$ is* maximal *(with respect to T), if it is not a proper prefix of another test in T.*

The execution of a test implies the execution of all its proper prefixes. Thus, to execute a test suite only its maximal tests have to be considered. As tests should be applied in the initial states, the implementation must be brought to its initial state before the application of a test. If the test suite possesses only a single maximal test, this can be accomplished by using a homing sequence, as in [8], [9], [18]. On the other hand, to execute a test suite with more than one maximal test, it is assumed that the implementation has a reset which reliably brings the machine to its initial state prior to applying the next test, e.g., [1], [2], [4], [14]. For the sake of simplicity, we define the length of a test α as $|\alpha| + 1$, i.e., the number of inputs plus a reset needed to bring the machine to the initial state, regardless of the fact that test suites with a single maximal test does not actually require such a reset. The length of a test suite T is defined as the sum of the lengths of all its maximal tests.

Given a reduced FSM M with n states, let $\Im(M)$ be the set of all reduced complete deterministic FSMs with the same input alphabet and at most n states.

Definition 3. *A given test suite T of FSM M is* n-complete, *if for each FSM $N \in \Im(M)$, such that N and M are distinguishable, there exists a test in T that distinguishes them.*

If an n-complete test suite is the set of all prefixes of a single sequence $R \in \Omega_M$, i.e., R is its only maximal test, then

R is, in fact, a so-called "checking sequence," used for testing FSM without a reset operation [18].

In this paper, we are concerned with conditions that are sufficient to guarantee that a given test suite is n-complete. We first introduce the notion of confirmed sets of defined input sequences. Throughout this paper, let $N = (Q, q_0, I, O', D', \Delta, \Lambda)$, where $D' = Q \times I$, be an arbitrary element of $\Im(M)$. Given a test suite T, let $\Im_T(M)$ be the set of all $N \in \Im(M)$, such that N and M are T-equivalent.

Definition 4. *Let T be a test suite of an FSM $M = (S, s_0, I, O, D, \delta, \lambda)$ and $K \subseteq T$. The set K is* confirmed *if $\delta(s_0, K) = S$ and, for each $N \in \Im_T(M)$, it holds that for all $\alpha, \beta \in K$, $\Delta(q_0, \alpha) = \Delta(q_0, \beta)$ if and only if $\delta(s_0, \alpha) = \delta(s_0, \beta)$. An input sequence is* confirmed *if there exists a confirmed set that contains it.*

In words, a confirmed set of input sequences contains transfer sequences for all states of M and any sequences converging (i.e., leading to a same state) in any FSM that has the same output responses to T and has as many states as M also converge in M. This key property is exploited by methods for constructing complete test suites, such as [1], [2], [4], [5], [9], [10], [8], [15], [17], [18], [20], in one way or another.

Notice that, according to Definition 4, we can establish that two sequences in a confirmed set for a given test suite T converge in any FSM that reacts to T as the FSM M only by determining that they converge in the FSM M.

The next theorem states that, for a given test suite to be n-complete for given FSM, it suffices that there exists a confirmed set which contains the empty sequence and covers each transition of FSM. A set of input sequences *covers* a transition if the set contains a transfer sequence for its initial state and the sequence is extended in the confirmed set with the input labelling the transition.

Theorem 1 (Sufficient Conditions for n-Completeness of a Test Suite). *Let T be a test suite of an initially connected reduced FSM $M = (S, s_0, I, O, D, \delta, \lambda)$ with n states. T is n-*complete *for M, if there exists a confirmed set $K \subseteq T$ with the following properties:*

1. $\varepsilon \in K$.
2. *For each $(s, x) \in D$, there exist $\alpha, \alpha x \in K$, such that $\delta(s_0, \alpha) = s$.*

Proof. Let $N \in \Im_T(M)$. As M is initially connected, for each $s \in S$, there exists $\alpha \in K$, such that $\delta(s_0, \alpha) = s$. For each $\beta \in K$, if $\delta(s_0, \beta) \neq \delta(s_0, \alpha)$, then we have that $\Delta(q_0, \beta) \neq \Delta(q_0, \alpha)$. Thus, $|Q| = n$. Consequently, there exists a bijection $f : S \to Q$, such that for each $\alpha \in K, f(\delta(s_0, \alpha)) = \Delta(q_0, \alpha)$. As $\varepsilon \in K, f(s_0) = q_0$. We prove that, for each $\nu \in \Omega_M, f(\delta(s_0, \nu)) = \Delta(q_0, \nu)$ using induction on ν, and, moreover, $\lambda(s, x) = \Lambda(f(s), x)$, for each $(s, x) \in D$.

If $\nu = \varepsilon$, we have $\nu \in K$, and, by definition, $f(\delta(s_0, \nu)) = \Delta(q_0, \nu)$. Let $\nu = \varphi x$ and assume that $f(\delta(s_0, \varphi)) = \Delta(q_0, \varphi)$. There exists $\alpha \in K$, such that $\delta(s_0, \alpha) = \delta(s_0, \varphi)$ and $\alpha x \in K$. Thus, we have that $f(\delta(s_0, \alpha x)) = \Delta(q_0, \alpha x)$ and $\Delta(q_0, \alpha) = f(\delta(s_0, \alpha)) = f(\delta(s_0, \varphi)) = \Delta(q_0, \varphi)$. It follows that

$$\begin{aligned} f(\delta(s_0, \varphi x)) &= f(\delta(\delta(s_0, \varphi), x)) = f(\delta(\delta(s_0, \alpha), x)) \\ &= f(\delta(s_0, \alpha x)) = \Delta(q_0, \alpha x) = \Delta(\Delta(q_0, \alpha), x) \\ &= \Delta(\Delta(q_0, \varphi), x) = \Delta(q_0, \varphi x). \end{aligned}$$

Therefore, $f(\delta(s_0, \varphi x)) = \Delta(q_0, \varphi x)$ and, by induction, for any $\nu \in \Omega_M, f(\delta(s_0, \nu)) = \Delta(q_0, \nu)$.

For each $(s, x) \in D$, there exists $\alpha x \in T$, $\delta(s_0, \alpha) = s$, $\alpha \in K$. Therefore, $\lambda(\delta(s_0, \alpha), x) = \Lambda(\Delta(q_0, \alpha), x)$. As $\alpha \in K$, we have that $\Delta(q_0, \alpha) = f(s)$ and, as N is T-equivalent to M, it follows that $\lambda(s, x) = \Lambda(f(s), x)$.

Suppose finally that N can be distinguished from M. Therefore, there exists a defined sequence $\nu x \in \Omega_M$, such that $\lambda(s_0, \nu) = \Lambda(q_0, \nu)$ and $\lambda(s_0, \nu x) \neq \Delta(q_0, \nu x)$. There exist $\alpha \in K$, such that $\delta(s_0, \alpha) = \delta(s_0, \nu)$, and $\alpha x \in K$, such that $\lambda(\delta(s_0, \alpha), x) = \Lambda(f(\delta(s_0, \alpha)), x)$. From $\delta(s_0, \alpha) = \delta(s_0, \nu)$, it follows that $\lambda(\delta(s_0, \nu), x) = \Lambda(f(\delta(s_0, \nu)), x) = \Lambda(\Delta(q_0, \nu), x)$; and from $\lambda(s_0, \nu) = \Lambda(q_0, \nu)$, it follows that

$$\begin{aligned} \lambda(s_0, \nu x) &= \lambda(s_0, \nu)\lambda(\delta(s_0, \nu), x) = \Lambda(q_0, \nu)\Lambda(\Delta(q_0, \nu), x) \\ &= \Lambda(q_0, \nu x). \end{aligned}$$

The resulting contradiction concludes the proof. □

If all the sequences in T are prefixes of a single input sequence, the test can be applied without a reliable reset. Thus, the conditions apply to both testing scenarios, with and without a reliable reset operation. In Section 5, we show that they are weaker than those known in the literature in either case.

The following lemmas indicate several possibilities for constructing a confirmed set. Our first lemma presents a sufficient condition for a minimal state cover (which contains a single transfer sequence for each state) to be a confirmed set. Given a test suite T of an FSM M, two sequences $\alpha, \beta \in T$ are *T-distinguishable* (or simply distinguishable), if there exist $\alpha\gamma, \beta\gamma \in T$, such that $\lambda(\delta(s_0, \alpha), \gamma) \neq \lambda(\delta(s_0, \beta), \gamma)$.

Lemma 1. *Let T be a test suite of FSM M and K be a minimal state cover. If each two sequences of K are T-distinguishable, then K is confirmed.*

Proof. Let $N \in \Im_T(M)$. The set K contains exactly n transfer sequences for all states of M, then, for each $s \in S$, there exists only one sequence in K that takes M to s. For any $\alpha, \beta \in K$, we have that $\delta(s_0, \alpha) \neq \delta(s_0, \beta)$ and $\Delta(q_0, \alpha) \neq \Delta(q_0, \beta)$. Therefore, as N has no more states than M, we have that $|\Delta(q_0, K)| = n$, and K is confirmed. □

The next statements indicate sufficient conditions for adding a sequence to a set while preserving the property of "being confirmed" of the set, based on which confirmed sets can incrementally be derived.

Lemma 2. *Let K be a confirmed set and α be a transfer sequence for state s. If for each $s' \in S \setminus \{s\}$, there exists $\beta \in K$, $\delta(s_0, \beta) = s'$, such that α and β are T-distinguishable, then the set $K \cup \{\alpha\}$ is confirmed.*

Proof. Let $N \in \Im_T(M)$. Let $f : S \to Q$ be a bijection, such that for each $\chi \in K, f(\delta(s_0, \chi)) = \Delta(q_0, \chi)$. It is sufficient to show that $f(s) = \Delta(q_0, \alpha)$. For each $s' \in S \setminus \{s\}$, there exists $\beta \in K, \delta(s_0, \beta) = s'$, such that α and β are T-distinguishable. Therefore, we have that $\Delta(q_0, \alpha) \neq \Delta(q_0, \beta) = f(s')$. It follows that $\Delta(q_0, \alpha) = f(s)$ and, thus, $K \cup \{\alpha\}$ is confirmed. □

The next statement relies on the fact that, if proper prefixes of some transfer sequences converge, then the sequences converge as well.

Lemma 3. *Let K be a confirmed set and $\alpha \in T$. If there exist $\beta, \chi \in K$, such that $\delta(s_0, \beta) = \delta(s_0, \chi)$, and a sequence φ, such that $\beta\varphi \in K$ and $\chi\varphi = \alpha$, then the set $K \cup \{\alpha\}$ is also confirmed.*

Proof. Let $N \in \Im_T(M)$. As χ, β and $\beta\varphi$ are in K, we have that $\Delta(q_0, \chi) = \Delta(q_0, \beta)$ and, therefore, it follows that

$$\begin{aligned}\Delta(q_0, \beta\varphi) &= \Delta(\Delta(q_0, \beta), \varphi) = \Delta(\Delta(q_0, \chi), \varphi) = \Delta(q_0, \chi\varphi) \\ &= \Delta(q_0, \alpha).\end{aligned}$$

Thus, as $\delta(s_0, \beta\varphi) = \delta(s_0, \alpha)$ and $\Delta(q_0, \beta\varphi) = \Delta(q_0, \alpha)$, we have that $K \cup \{\alpha\}$ is confirmed. □

In the following theorem, we summarize the above lemmas in sufficient conditions for a given set of defined input sequences to be confirmed:

Theorem 2 (Sufficient Conditions for the Existence of a Confirmed Set). *Let T be a test suite of FSM M with n states and $L \subseteq T$ be a set of k sequences, $k \geq n$. For an arbitrary ordering of the sequences $\alpha_1, \ldots, \alpha_k$ in L, let $L_i = \{\alpha_j \in L \mid 1 \leq j \leq i\}$. Then L is a confirmed set if there exists an ordering $\alpha_1, \ldots, \alpha_k$, such that the corresponding $L_1, \ldots, L_k$ satisfy the following conditions:*

1. *L_n is a minimal state cover such that every two sequences are T-distinguishable.*
2. *If $k > n$, then for each $\alpha_i, n < i \leq k$, it holds that either*
   a. *for each $s \in S \setminus \{\delta(s_0, \alpha_i)\}$, there exists $\beta \in L_{i-1}, \delta(s_0, \beta) = s$, such that α_i and β are T-distinguishable; or*
   b. *there exist χ, β, and φ, such that $\alpha_i = \chi\varphi$, and $\beta\varphi$, β, $\chi \in L_{i-1}$, $\delta(s_0, \beta) = \delta(s_0, \chi)$;*

Proof. We prove by induction on L_i. For the basis step, L_n is a confirmed set by Lemma 1. For the induction step, assume that L_i, $n \leq i < k$, is a confirmed set. We show that L_{i+1} is also confirmed. If 2a holds, then Lemma 2 applies; otherwise, if 2b holds, Lemma 3 does. Consequently, the set $L_i \cup \{\alpha_i\} = L_{i+1}$ is confirmed. □

4 Algorithm for Checking n-Completeness

In this section, we present an algorithm for determining the n-completeness of a given test suite based on Theorems 1 and 2. As the conditions of these theorems are sufficient, if the algorithm terminates with a positive result, then the test suite is indeed n-complete. However, as the conditions are not necessary, based on a negative answer, we cannot conclude that the test suite is not n-complete.

The algorithm involves three main steps:

1. minimal confirmed sets are identified by applying Lemma 1 to a given test suite T;
2. the minimal confirmed sets are repeatedly extended by the application of Lemmas 2 and 3 to sequences of T as long as possible, thus obtaining maximal confirmed sets; and
3. the maximal confirmed set are checked for satisfaction of Theorem 1.

We first apply Lemma 1 to find minimal confirmed sets (i.e., containing a single transfer sequence for each state of M), which are subsets of T with n pairwise T-distinguishable sequences. The problem of finding minimal confirmed sets can be cast as a problem of finding cliques in a graph, as follows: We define a *distinguishability graph* G on T as a graph whose vertices are the sequences in T, such that two vertices are adjacent in G if and only if the corresponding sequences are T-distinguishable. Then, the sequences that appear in a clique of size n (an n-clique) of G form a confirmed set. The problem of finding n-cliques in an arbitrary graph is NP-complete [11]. However, several properties of distinguishability graphs can be used to formulate heuristics which allow dealing with large graphs. Notice first that G is an n-partite graph, since the sequences that transfer to same state are not adjacent and, therefore, we can partition its vertices into n blocks. Thus, we deal with the special case of finding n-cliques in an n-partite graph. This problem has already been investigated in [7], where a specialized algorithm is proposed to find all n-cliques. The algorithm implements a branch-and-bound approach, where a partial solution is extended in a search tree (branching), and the search is pruned as soon as it is possible to determine that a given partial solution is fruitless (bounding). The initial partial solution is a trivial empty clique. It is extended with sequences that are adjacent to every sequence in the partial clique. Based on the fact that the graph is n-partite, the authors propose heuristics that help determine very early when a partial clique cannot be extended to an n-clique. The proposed heuristics are also useful to solve our problem. Moreover, differently from that work, we do not need to find all n-cliques, as discussed below.

From a minimal confirmed set K, we can obtain a confirmed set $K' \subseteq T$, such that $K \subseteq K'$ and K' is the largest set which satisfies the conditions in Theorem 2. To determine K', we initialize a set K_{cur} (a current confirmed set) with K. Then, we iteratively select a sequence $\alpha \in T \setminus K_{cur}$ and try to apply either Lemma 2 or Lemma 3. If no new sequence satisfies them, the confirmed set K_{cur} so far obtained is the largest one.

Notice that it is not necessary to check a minimal confirmed set K that is included in some largest confirmed set K' that was already analyzed, as stated in the next lemma.

Lemma 4. *Let K be a largest confirmed set that satisfies the conditions of Theorem 2. Let K' be a minimal confirmed set and K'' be the largest confirmed set obtained by applying Lemmas 2 and 3 to the set K'. Then if $K' \subseteq K$, it holds that $K'' \subseteq K$.*

Proof. We prove by contradiction. Assume that $K' \subseteq K$, and $K'' \not\subseteq K$. The sequences of K'' can be ordered as $\alpha_1, \ldots, \alpha_k$, according to Theorem 2. Let j be such that $K_j = \{\alpha_1, \ldots, \alpha_{j-1}\} \subseteq K$, but $\alpha_j \notin K$. Thus, there exists a set of sequences $W \subseteq K_j$ which, in conjunction with α_j, satisfy the conditions of either Lemma 2 or Lemma 3. In this case, K can be extended by the inclusion of α_j, since $W \subseteq K$. However, this contradicts the fact that K is a largest set with respect to the conditions of Theorem 2. □

Thus, according to Lemma 4, after finding an n-clique that represents a minimal confirmed set, the search tree can be bounded whenever it can be concluded that any n-clique obtained from a given partial clique would be included in

some largest confirmed set already determined. As discussed in Section 6, this heuristic is a key factor for the performance of the algorithm.

It remains to verify whether the obtained maximal confirmed set satisfies Theorem 1, which is a straightforward step. If it does, the test suite is n-complete for M. Otherwise, if another minimal confirmed set can be found, the whole process iterates again.

We finally present the algorithm in details.

Algorithm 1.
Input: An FSM M and a test suite T.
Output: **True**, if T is n-complete according to Theorems 1 and 2.

1. Build the distinguishability graph G of T.
2. Let L be the empty set.
3. Determine (by using the branch-and-bound approach) an n-clique K of G, such that there does not exist $K' \in L$ with $K \subseteq K'$. If no such a clique exists, then terminate with the answer **False**.
4. Find a sequence $\alpha \in T \setminus K$, such that either Lemma 2 or Lemma 3 can be applied. If no such a sequence exists, go to Step 6.
5. Include α in K and go to Step 4.
6. If K satisfies Theorem 1, then terminate with the answer **True**.
7. Include K in L and go to Step 3.

We have implemented Algorithm 1 in a tool and applied the tool to randomly generated FSMs and test suites. The obtained results are discussed in Section 6.

5 Comparison with Previous Work

In this section, we show that the sufficient conditions in [2] and [18] are special cases of the conditions presented in this paper. The conditions of [18] are formulated for checking sequences; moreover, they can only be applied to strongly connected FSMs possessing a diagnostic sequence, which may not exist for an arbitrary reduced FSM. The conditions of [2] apply to initially connected FSMs even without diagnostic sequences; however, an implementation is assumed to have a reliable reset operation and, thus, test suites can have more than one maximal test.

In [2], the authors present the weakest sufficient conditions found in literature for an n-complete test suite using the reset operation. The conditions are stated in Theorem 3, slightly rephrased using our notations.

Theorem 3 [2]. *Let T be a test suite and Q be a prefix-closed state cover of an FSM M, such that the following conditions hold:*

1. *For all sequences $\alpha, \beta \in Q$, such that $\delta(s_0, \alpha) \neq \delta(s_0, \beta)$, it holds that α and β are T-distinguishable.*
2. *For each defined transition $(s, x) \in D$, there exists $\alpha x \in T$, such that $\delta(s_0, \alpha) = s$, with the following properties.*
   a. *For each $\beta \in Q$, such that $\delta(s_0, \beta) \neq s$, it holds that α and β are T-distinguishable.*
   b. *For each $\beta \in Q$, such that $\delta(s_0, \beta) \neq \delta(s, x)$, it holds that αx and β are T-distinguishable.*

Then, T is n-complete.

We show that Theorem 3 is a special case of Theorem 1.

Theorem 4. *Let T be a test suite as in Theorem 3. Then T satisfies the conditions of Theorem 1.*

Proof. We first show that a state cover $Q \subseteq T$ defined in Theorem 3 is a confirmed set. Let $Q_0 \subseteq Q$ be a minimal state cover. Clearly, each two sequences in Q_0 are T-distinguishable, by Condition 1. Then, by Lemma 1, Q_0 is a confirmed set. Let $\nu \in Q \setminus Q_0$. By Condition 1, ν is T-distinguishable from each sequence $\alpha \in Q_0$ which does not transfer to the same state as ν. Consequently, by Lemma 2, we have that $Q_0 \cup \{\nu\}$ is a confirmed set and so is Q. Condition 2.i implies that $Q \cup \{\alpha\}$ is a confirmed set, since α is T-distinguishable from each sequence $\beta \in Q$ which does not transfer to the same state as α and, therefore, Lemma 2 can be applied. Similarly, $Q \cup \{\alpha x\}$ is a confirmed set, as αx is T-distinguishable from each sequence $\beta \in Q$ which does not lead to the same state as αx. Thus, let K be a confirmed set which includes Q and the corresponding sequences α and αx, for each defined transition $(s, x) \in D_M, \delta(s_0, \alpha) = s$. Therefore, as $\varepsilon \in Q$ (recall that Q is prefix-closed), K satisfies the conditions of Theorem 1. □

We now demonstrate that the converse is not true, by showing an n-complete test suite for which Theorem 3 does not hold, while Theorem 1 does. Consider the FSM in Fig. 1 and the test suite T_1 which contains $xyyxy$, $yyyyyyxyyy$, and their prefixes. It does not satisfy the conditions of Theorem 3, since there is no state cover in the test suite T_1 with the required characteristics. Indeed, xyy is the only sequence which leads to state 2 and is followed by x in T_1. Therefore, $\alpha = xyy$ is the only sequence that could be used in Condition 2 for the defined transition $(2, x)$. However, the input sequence xy is the only sequence applied after the sequence xyy, but it does not distinguish state 2 from state 3, since input x is not defined in latter state. Thus, Condition 2i is violated.

Nonetheless, by using Lemmas 1, 2, and 3, we can find a confirmed set, which satisfies the conditions of Theorem 1. We have that the set $\{\varepsilon, y, yy, yyyyyyx\} = K_0$ is confirmed, by Lemma 1. By repeatedly applying Lemma 2, we can prove that the set $K_0 \cup \{xyyx, yyy, yyyyyyxy, yyyyyyxyy\} = K_1$ is confirmed. After several applications of Lemma 3, we obtain the confirmed set $K_1 \cup \{yyyy, yyyyy, yyyyyy\} = K_2$. Now, we can apply Lemma 2 to prove that $K_2 \cup \{x\} = K_3$ is confirmed. Finally, we add sequences xy and xyy by applying Lemma 3 and obtain the confirmed set $\{\varepsilon, x, xy, xyy, xyyx, y, yy, yyy, yyyy, yyyyy, yyyyyy, yyyyyyx, yyyyyyxy\}$, which satisfies the conditions of Theorem 1.

Ural et al. [18] present conditions for a sequence to be a checking sequence. In that work, a sequence is defined to be a checking sequence if it can distinguish a complete strongly connected deterministic reduced FSM M from each FSM with at most as many as states as M that is not isomorphic to M. The conditions rely on the existence of a diagnostic sequence (also called a distinguishing sequence). We first restate a definition used in [9], [10], [18] for constructing checking sequences for complete as well as partial reduced FSMs.

Definition 5. *Given $R \in \Omega_M$, let d be a diagnostic sequence of a strongly connected deterministic reduced (possibly partial) FSM M. Then,*

1. *A prefix α of R is (d-)recognized in R if αd a prefix of R.*
2. *If α, β, and $\alpha\gamma$ are recognized in R and $\delta(s_0, \alpha) = \delta(s_0, \beta)$, then $\beta\gamma$ is recognized in R.*
3. *If α and αx are recognized in R and $\delta(s_0, \alpha) = s$, then the transition (s, x) is verified in R.*

We present a theorem which is similar to the one formulated in [18], but is stronger in the sense that all the implementation FMSs which are distinguishable from the specification FSM are considered, and not only those which are not isomorphic [18] or not equivalent to the specification FSM [9], [10]. The statement is a special case of Theorem 1 and, thus, takes into account initialization faults as well, as opposed to [9], [10], [18]. Compared to the original version of the theorem, we add the requirement that d must be a prefix of the checking sequence.

Theorem 5. *Given $R \in \Omega_M$, if d is a prefix of R and every transition of M is verified in R, then the set of prefixes of R is an n-complete test suite.*

Proof. Let K_0 be the set of d-recognized prefixes of R. We first show that $\delta(s_0, K_0) = S$. Let $s \in S$. There exists at least one recognized sequence that leads to s, since every transition is verified and M is strongly connected. For a sequence to be recognized, either Condition 1 or Condition 2 must hold. For Condition 2, however, another recognized sequence that also leads to s is required and, consequently, at least one sequence satisfies Condition 1. Therefore, for each s, there exists at least one sequence that is d-recognized and, thus, $s \in \delta(s_0, K_0)$.

As d is a diagnostic sequence, for all $\alpha, \beta \in K_0$, such that $\delta(s_0, \alpha) \neq \delta(s_0, \beta)$, it holds that α and β are T-distinguishable. Then, by Lemma 1, K_0 is a confirmed set. Furthermore, we have that $\varepsilon \in K_0$, since d is a prefix of R. If $\alpha, \beta, \alpha\gamma$, and $\beta\gamma$ are prefixes of R and α, β, and $\alpha\gamma$ are in a confirmed set, then $\beta\gamma$ can also be included in the confirmed set, by Lemma 2. Consequently, if a sequence φ is recognized and K' is a confirmed set, then so is $K' \cup \{\varphi\}$. Let K be the set of all recognized prefixes of R. It follows that K is a confirmed set and $K_0 \subseteq K$. As every transition is verified in R, for each $(s, x) \in D$, there exist $\alpha, \alpha x \in K$. Therefore, by Theorem 1, the set of R's prefixes is n-complete. □

We now present an example of an n-complete test suite that satisfies Theorem 1, but not Theorem 5. Consider the FSM in Fig. 1 and the sequence $R = yyyyyyxyyyxyxyyxy$. The shortest diagnostic sequence for this FSM is yyy. The d-recognized sequences are ε, y, yy, yyy, and $yyyyyyx$. The recognized sequences are $yyyy$, $yyyyy$, and $yyyyyy$. Then, the set of verified transitions is $\{(1, y), (2, y), (4, y), (4, x)\}$, which includes only four out of seven defined transitions.

Now we demonstrate that the test suite T_2, containing the maximal test $R = yyyyyyxyyyxyxyyxy$, satisfies Theorem 1 and, thus, that R is a checking sequence. First, it holds, by Lemma 1, that $\{\varepsilon, y, yy, yyyyyyx\} = K_0$ is a confirmed set. By the application of Lemma 2, we have that $K_0 \cup \{yyy\} = K_1$ is confirmed. We repeatedly apply Lemma 3 to prove that $K_1 \cup \{yyyy, yyyyy, yyyyyy\} = K_2$ is a confirmed set. Using Lemma 2, we obtain the confirmed set $K_2 \cup \{yyyyyyxy, yyyyyyxyy\} = K_3$. Then, $K_3 \cup \{yyyyyyxyyy, yyyyyyxyyyx, yyyyyyxyyyxy\} = K_4$ is a confirmed set, according to Lemma 3. Next, we have that $K_4 \cup \{yyyyyyxyyyxyx\} = K_5$ is also confirmed (Lemma 2). Now, we can prove that $K_5 \cup \{yyyyyyxyyyxyxy, yyyyyyxyyyxyxyy\} = K_6$ is a confirmed set. Finally, the sequences *yyyyyyxyyyxyxyyx* and *yyyyyyxyyyxyxyyxy* are also confirmed according to Lemmas 2 and 3, respectively. The resulting confirmed set satisfies the conditions of Theorem 1.

Another interesting feature of the conditions is that they are more flexible than the previous ones. For instance, although both the test suites T_1 and T_2 satisfy conditions proposed in this paper, they do not satisfy the conditions of [2] and [18]. The test suite T_1 has length 17 and two maximal tests, whereas the test suite T_2 has length 18 and a single maximal test. Thus, the proposed conditions are not parameterized with the number of resets needed to execute all the tests; this feature allows to elaborate a test generation method to produce a test suite which is most suitable (in terms of the number of tests) to a given situation.

Another approach to determine whether a given test suite is n-complete is presented in [19], [14]. Given an FSM M and a test suite T, the tree machine with the set of defined sequences being exactly T is first constructed. Then one needs to construct all the possible reduced forms of the tree machine (the FSM M is one of them), using an existing algorithm for partial FSM minimization (recent publications on this topic include, e.g., [6], [13]). If at least one of the obtained reduced FSMs is distinguishable from M, then T is not n-complete. Otherwise, it is n-complete. Compared to our approach, this method is exhaustive, while ours is approximate, in the sense that we can positively identify some n-complete test suites, but cannot provide definitive negative answer. However, the problem of partial FSM minimization is NP-complete and the existing algorithms can deal only with small machines and small test suites, as the experimental results of recent publications (e.g., [6]) show. Our method must also deal with the NP-complete problem of finding an n-clique. Nonetheless, the heuristics derived from Lemma 4 and the fact that the distinguishability graph is n-partite allow us to cope with significantly larger FSMs and test suites (compared to [6], [19]), as the experimental results in Section 6 indicate. On the other hand, the solution of [6] requires that all n-cliques be found and checked, so its applicability is reduced to FSMs with few states.

6 Experimental Results

To evaluate the proposed sufficient conditions as well as the method for checking test completeness a number of experiments involving random generation of FSMs and tests were performed using a tool, called `Chico` (**Ch**eck**i**ng **co**mpleteness), which checks whether test suites satisfy the conditions proposed in this paper. The first set of experiments addressed the scalability of the method, and the second compared it with the checkers of conditions of [2] and [18].

TABLE 1
Number of Minimal Confirmed Sets and Size of the First Found Largest Confirmed Sets

Number of States	Number of Minimal Confirmed Sets	Size of the First Found Largest Confirmed Set
4	3900	157
5	3888	172
6	12321	158
7	216178	184
8	1206465	192
9	620544	184
10	654750	182

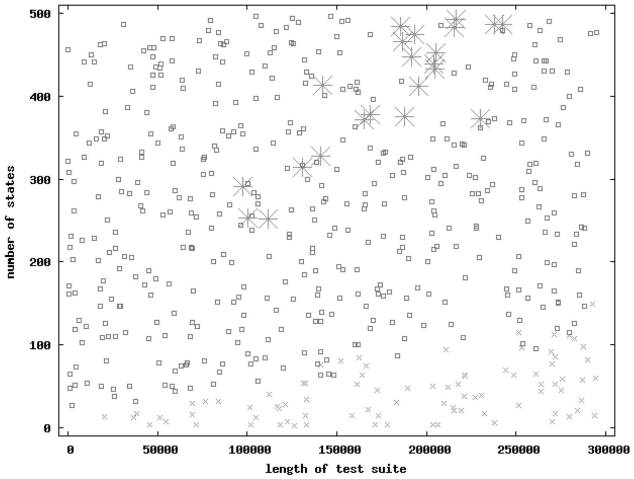


Fig. 2. Distribution of runs.

In the experiments, we used randomly generated FSMs and test suites. We randomly generate initially connected reduced FSMs in the following way. Sets of states, inputs, and outputs with the required number of elements are first created. The generation proceeds then in three phases. In the first phase, a state is selected as the initial state and marked as "reached." Then, for each state s not marked as "reached," the generator randomly selects a reached state s', an input x, and an output y and adds a transition from s' to s with input x and output y, and mark s as "reached." When this phase is completed, an initially connected FSM is obtained. In the second phase, the generator adds, if needed, more transitions (by randomly selecting two states, an input, and an output) to the machine until the required (given a priori) number of transitions is obtained. In the third phase, the distinguishability of each pair of distinct states is checked. If the FSM is not reduced, it is discarded and another FSM is generated.

Once a reduced FSM is obtained, a test suite is randomly generated as follows: We start with a test suite T_{cur} containing only the empty sequence, i.e., $T_{cur} = \{\varepsilon\}$. Then, a defined sequence α is iteratively generated starting from $\alpha = \varepsilon$ by adding to it an input randomly selected among those defined in the state reached by the current sequence. The sequence growing process terminates as soon as $\alpha \notin T_{cur}$; the sequence α is then included into T_{cur}. After the inclusion of α, the number of sequences in T_{cur} is increased by one.

6.1 Scalability of the Proposed Algorithm

An important question is how many minimal confirmed sets have to be analyzed for a given test suite. To answer this question, we executed `Chico` with the FSM in Fig. 1 and 10,000 randomly generated test suites. We observed that the tool usually finds the first minimal confirmed set rather quickly and the maximal confirmed set is then determined. The subsequent search for another minimal confirmed set is bounded quickly due to Lemma 4. In this experiment, no test suite required the analysis of more than two minimal confirmed sets, and in most cases, only a single minimal confirmed set was analyzed. Moreover, only in 144 out of 10,000 test suites, two minimal confirmed sets were used. This experiment indicates that the number of minimal confirmed sets to be analyzed may not be always large in spite of the fact that their total number grows exponentially with the number of states. This dependency is an essential impediment to any approach explicitly enumerating all n-cliques of a graph, e.g., [19]. However, for our algorithm, the larger the number of n-cliques, the easier it is to find one of them and the remaining search can be bounded early. Table 1 illustrates the saving due to Lemma 4 in another set of experiments. We randomly generated reduced complete FSMs with two inputs, two outputs, and test suites of with 200 tests and selected the FSMs for which the number of minimal confirmed sets is the largest, representing a worst-case scenario. For none of them, the test suite was determined to be n-complete by the tool. Indeed, the number of minimal confirmed sets is large (see, for instance, the experiments with the FSM with eight states). However, the size of the largest confirmed set obtained from the first identified minimal confirmed set is also large. Then, all other minimal confirmed sets are included in the first largest confirmed set and this fact can be established rather early, bounding the search.

During some of the experiments with large FSMs and tests, the runtime to find the first minimal confirmed set becomes unacceptably long. This is not surprising, since the problem is NP-complete and even with the heuristics employed in the tool it may eventually take an exponential amount of time to find a minimal confirmed set. An important question here is how often the tool fails due to the impossibility of finding a minimal confirmed set in a reasonable amount of time. We have chosen a timeout of one hour to terminate executions. All the experiments were run on a Pentium IV HT 64 bits 3.4 GHz computer, with 2 Gb of memory. We generated 500 FSMs with 10 inputs, 10 outputs, number of states randomly chosen between one and 500 as well as 500 test suites of length between one and 300,000. Fig. 2 shows the results, where small crosses represent runs that ended before the timeout expiration with a positive answer (the test suite was n-complete), small squares represent runs that ended before the timeout with a negative answer, and big stars represent the ones lasting at least one hour. There were 22 runs terminated by the timeout, which correspond to 4.4 percent of the executions;

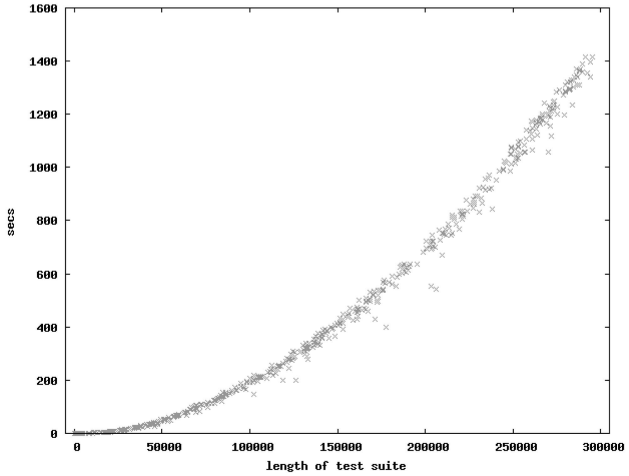


Fig. 3. Execution time variation with the length of a test suite.

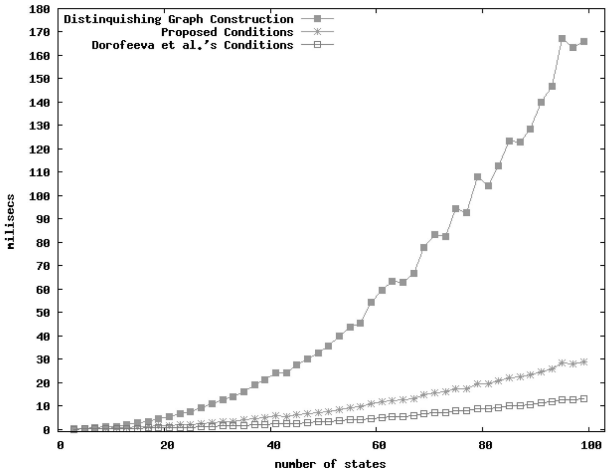


Fig. 4. Average execution time for checking the proposed conditions and the conditions from [2] for complete test suites.

none of them occurred for FSMs with fewer than 200 states or for test suites with length smaller than 80,000.

To see how execution time depends on the number of states of the FSMs, we randomly generated 500 reduced complete FSMs with 10 inputs, 10 outputs, and states ranging from 3 to 500, as well as test suites with 20,000 tests. We consider only the runs that were not ended by a timeout. As seen in the previous experiment, the probability of a run ending by timeout for this setting is negligible, since timeouts only occurred with larger test suites. The average time was 61.046 seconds and the standard deviation was 3.451 seconds. Thus, all things being equal, the execution time varies only slightly with the number of states. Actually, we observed that the parameter with the greatest impact on execution time is the test suite length, as discussed next.

Fig. 3 shows how the execution time grows as the test suite length increases. We generated 500 complete FSMs with 10 inputs, 10 outputs, and the number of states ranging from 3 to 500. The length of the test suites ranges from 1 to 300,000. Since the number of edges in the distinguishability graph and, consequently, the time for constructing it, grows quadratically with the test suite length, the overall execution time increases in the same way. We notice that even for test suites of length as big as 300,000 and for FSMs with up to 500 states, the tool was able to produce a result in less than 1,500 seconds. In this experiment, we also excluded the runs in which the tool was terminated by timeout. For larger test suites, the tool runs out of memory, since the amount of memory required for data structures used to build and represent the distinguishability graph also grows quadratically with the length of the test suite.

6.2 Experimental Comparison with Previous Work

The conditions proposed in this paper are more complex than the conditions in previous work, except for [19]. Therefore, an important question is what is the overhead of their checking. We compare the scalability of methods checking the proposed and existing sufficient conditions. Notice that neither [2] nor [18] discuss how the conditions can be checked, since n-complete test suite generation is the focus of either work. Nonetheless, checkers for both conditions could easily be derived from the tool `Chico`, by limiting the application of certain lemmas and heuristics.

For Dorofeeva et al.'s conditions, Lemmas 3 and 4 are not applicable and, moreover, the use of Lemma 2 is limited to prefix-closed confirmed sets. We implemented a checker of the conditions of [2], named `ChicoD`, and determined the time required to check the n-completeness of test suites in two scenarios.

In the first scenario, complete reduced FSMs are randomly generated, but the test suites are obtained by the State Counting method [15], which produces n-complete test suites. This scenario is the most favorable for the conditions of [2], since a test suite obtained by the above method satisfies them and, moreover, the determination of a suitable state cover required by those conditions is straightforward. Not surprisingly, all things being equal, `Chico` needs more time than `ChicoD`. In Fig. 4, we present the average execution time for 100 FSMs with three inputs, three outputs, and number of states ranging from 3 to 100, totalling 9,800 FSMs. For each FSM, we generated an n-complete test suite using State Counting method. Notice that the length of the test suite increases as the number of states increases. We divided the execution time into two parts: the time required to construct the distinguishing graph of a given test suite and the time for checking the respective conditions based on the graph. The results show that, although `Chico` employs more complex conditions, the overhead is still reasonable, even when the conditions of [2] can be more promptly checked. Notice that as the FSMs grow (and, consequently, the length of the test suite increases), the time required to construct the distinguishing graph increases faster than the time required to check the conditions. The distinguishing graph allows one to avoid recalculating the T-distinguishability of pairs of sequences and, thus, cannot be removed without an increase in the execution time. We, thus, observe that the overhead tends to become insignificant, even in the scenario that is most favorable to Dorofeeva et al.'s conditions.

In the second scenario, we used randomly generated test suites. In this scenario, the heuristics that we implemented in `Chico` allow treating much larger FSMs than the checker

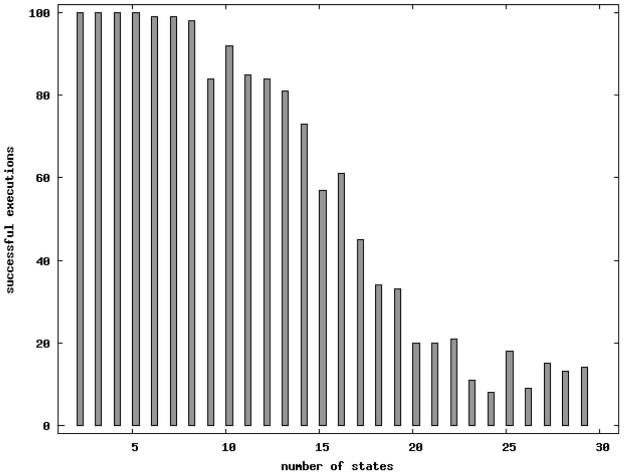


Fig. 5. Percentage of executions of ChicoD which terminate before timeout for random test suites.

of Dorofeeva et al.'s conditions. We randomly generated 100 complete reduced FSMs with three inputs, three outputs, and number of states from 3 to 30. The test suites are randomly generated with 1,000 test cases. Each execution of `Chico` terminated in less than 0.2 seconds. On the other hand, the execution of `ChicoD` took often more than 10 minutes, which is a timeout we set to terminate executions. Fig. 5 shows the number of executions that terminate before the timeout. Notice that while for small FSMs the conditions of Theorem 3 could be checked for every FSM, the probability that `ChicoD` fails to verify the Dorofeeva et al.'s conditions within a reasonable amount of time grows with the state number of a specification FSM, since Lemma 4 is not applicable and state covers to verify are numerous.

To check whether a given input sequence satisfies Theorem 5, it is first necessary to determine if any of its prefixes is a diagnostic sequence. Then, only this diagnostic sequence is considered to determine T-distinguishability of sequences. However, the actual limitation of those conditions is their applicability. For instance, experimental studies [3] indicate that the probability of a randomly generated FSM to have a diagnostic sequence is low, circa 15 percent. Moreover, even if an FSM has a diagnostic sequence, the probability that an input sequence satisfies the conditions of Theorem 5 is yet smaller. We implemented a checker of the conditions of [18], named `ChicoU`. Lemmas 1, 2, and 3 are still applicable in checking those conditions, but Lemma 4 is not. We generated 100 complete reduced FSMs with three inputs, three outputs, and number of states from 3 to 15. Then, we randomly generated input sequences of length 1,000. The test suite obtained from each input sequence is then checked with `Chico`. If Theorem 1 is not satisfied, the FSM and test suite are discarded and others are generated. We repeat this process until we obtain a set of 1,200 FSMs (i.e., 100 for each size of FSMs) and respective checking sequences that satisfy Theorem 1. Then, using `ChicoU`, we checked whether they also satisfy Theorem 5. We observed that the number of checking sequences satisfying Theorem 5 drops quickly. For FSMs with three states, 88 out of 100 sequences satisfy it, while for FSMs with eight states only two sequences do. For bigger FSMs, no checking sequence satisfies Theorem 5.

The experimental results obtained for relatively large FSMs and tests indicate that the proposed conditions have a wider applicability compared to [18]; checking them scales better than checking the conditions [19] and [2].

7 Conclusions

In this paper, we presented sufficient conditions for test suite n-completeness that are weaker than those known in the literature. The conditions apply to both testing scenarios, with and without reliable reset operation. They can be used in several ways. On one hand, sufficient conditions can guide the definition of new generation methods or the improvement of existing ones. Elaboration of such a method based on the proposed sufficient conditions is an open research issue. On the other hand, the n-completeness of existing test suites can be checked by the algorithm we proposed. Strategies for minimizing complete tests without losing fault detection capability can also be elaborated. Although the algorithm requires the identification of a clique in a graph, an NP-complete problem, the experimental results we presented show that the algorithm can be used for relatively large FSMs and test suites.

As future work, we can mention several possible extensions of the presented results. First, it is interesting to see how Theorem 1 can be extended to the case of m-completeness, where $m \geq n$. Another possible generalization of conditions would be to consider nondeterministic specification FSMs. Finally, since the proposed test completeness conditions are only claimed to be sufficient, we believe that the quest for necessary and sufficient conditions will go on.

Acknowledgments

This work was in part supported by the Natural Sciences and Engineering Research Council of Canada under discovery grant OGP0194381 and by Brazilian Funding Agency CNPq under grant 200032/2008-9. The authors wish to thank the reviewers for their useful comments.

References

[1] T.S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. Software Eng.*, vol. 4, no. 3, pp. 178-187, May 1978.

[2] R. Dorofeeva, K. El-Fakih, and N. Yevtushenko, "An Improved Conformance Testing Method," *Formal Techniques for Networked and Distributed Systems*, pp. 204-218, Springer, 2005.

[3] R. Dorofeeva, N. Yevtushenko, K. El-Fakih, and A.R. Cavalli, "Experimental Evaluation of FSM-Based Testing Methods," *Proc. Third IEEE Int'l Conf. Software Eng. and Formal Methods*, pp. 23-32, 2005.

[4] S. Fujiwara, G.v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test Selection Based on Finite State Models," *IEEE Trans. Software Eng.*, vol. 17, no. 6, pp. 591-603, June 1991.

[5] G. Gonenc, "A Method for the Design of Fault Detection Experiments," *IEEE Trans. Computers*, vol. 19, no. 6, pp. 551-558, June 1970.

[6] S. Gören and F.J. Ferguson, "On State Reduction of Incompletely Specified Finite State Machines," *Computers & Electrical Eng.*, vol. 33, no. 1, pp. 58-69, 2007.

[7] T. Grunert, S. Irnich, H.-J. Zimmermann, M. Schneider, and B. Wulfhorst, "Finding All k-Cliques in k-Partite Graphs: An Application in Textile Engineering," *Computers & Operations Research*, vol. 29, pp. 13-31, 2002.

[8] F.C. Hennie, "Fault-Detecting Experiments for Sequential Circuits," *Proc. Fifth Ann. Symp. Circuit Theory and Logical Design,* pp. 95-110, 1964.
[9] R.M. Hierons and H. Ural, "Reduced Length Checking Sequences," *IEEE Trans. Computers,* vol. 51, no. 9, pp. 1111-1117, Sept. 2002.
[10] R.M. Hierons and H. Ural, "Optimizing the Length of Checking Sequences," *IEEE Trans. Computers,* vol. 55, no. 6, pp. 618-629, May 2006.
[11] R.M. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computations,* R.E. Miller and J.W. Thatcher, eds., pp. 85-103, Springer, 1972.
[12] E.P. Moore, "Gedanken-Experiments," *Automata Studies,* C. Shannon and J. McCarthy, eds., Princeton Univ. Press, 1956.
[13] J.M. Pena and A.L. Oliveira, "A New Algorithm for Exact Reduction of Incompletely Specified Finite State Machines," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 18, no. 11, pp. 1619-1632, Nov. 1999.
[14] A. Petrenko, G.v. Bochmann, and M. Yao, "On Fault Coverage of Tests for Finite State Specifications," *Computer Networks and ISDN Systems,* special issue on protocol testing, vol. 29, pp. 81-106, 1996.
[15] A. Petrenko and N. Yevtushenko, "Testing from Partial Deterministic FSM Specifications," *IEEE Trans. Computers,* vol. 54, no. 9, pp. 1154-1165, Sept. 2005.
[16] J.F. Poage and E.J. McCluskey Jr., "Derivation of Optimal Test Sequences for Sequential Machines," *Proc. IEEE Fifth Symp. Switching Circuits Theory and Logical Design,* pp. 121-132, 1964.
[17] M.P. Vasilevskii, "Failure Diagnosis of Automata," *Cybernetics,* vol. 4, pp. 653-665, 1973.
[18] H. Ural, X. Wu, and F. Zhang, "On Minimizing the Lengths of Checking Sequences," *IEEE Trans. Computers,* vol. 46, no. 1, pp. 93-99, Jan. 1997.
[19] M. Yao, A. Petrenko, and G.v. Bochmann, "Fault Coverage Analysis in Respect to an FSM Specification," *Proc. IEEE INFOCOM '94,* pp. 768-775, 1994.
[20] N. Yevtushenko and A. Petrenko, "Synthesis of Test Experiments in Some Classes of Automata," *Automatic Control and Computer Sciences,* vol. 24, no. 4, pp. 50-55, 1990.

Adenilso Simao received the BS degree in computer science from the State University of Maringa (UEM), Brazil, in 1998, and the MS and PhD degrees in computer science from the University of Sao Paulo (USP), Brazil, in 2000 and 2004, respectively. Since 2004, he has been a professor of computer science at the Computer System Department of USP. From August 2008 to July 2010, he has been on a sabbatical leave at Centre de Recherche Informatique de Montreal (CRIM), Canada. His research interests include software testing and formal methods. He is a member of the Brazilian Computer Society (SBC).

Alexandre Petrenko received the diploma degree in electrical and computer engineering from Riga Polytechnic Institute in 1970 and the PhD degree in computer science from the Institute of Electronics and Computer Science, Riga, USSR, in 1974. He was also awarded other degrees and titles, namely, "Doctor of Technical Sciences" and "Senior Research Fellow in Technical Cybernetics and Information Theory" from the Supreme Attestation Committee, Moscow, USSR, and "Doctor Habil. of Computer Science" from the Latvian Scientific Council, Riga, Latvia. Until 1992, he was head of a computer network research lab of the Institute of Electronics and Computer Science in Riga. From 1979 to 1982, he was with the Computer Network Task Force of the International Institute for Applied Systems Analysis (IIASA), Vienna, Austria. From 1992 to 1996, he was a visiting professor/researcher of the Universite de Montreal. He joined the Centre de Recherche Informatique de Montreal (CRIM) in 1996, where he is currently a senior researcher and team leader. In 2005, along with C. Campbell, M. Veanes, and J. Huo, he received the best paper award from the 17th IFIP International Conference on Testing of Communicating Systems. He has published more than 150 research papers and has given numerous invited lectures worldwide. He is a member of the IFIP TC6 Working Group 6.1 "Architectures and Protocols for Distributed Systems" and serves as a member of the program committee for a number of international conferences and workshops. He is a member of the steering committee of the IFIP International Conference on Testing of Communicating Systems (TestCom). His current research interests include formal methods and their application in distributed systems and computer networks.

Apêndice D

A. S. Simão, A. Petrenko. Fault Coverage-Driven Incremental Test Generation. Computer Journal, v. 53, p. 1508-1522, 2010

Advance Access publication on August 12, 2009 doi:10.1093/comjnl/bxp073

Fault Coverage-Driven Incremental Test Generation

ADENILSO SIMÃO[1,2,*] AND ALEXANDRE PETRENKO[2]

[1]*Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, São Paulo, Brazil*
[2]*Centre de recherche informatique de Montreal (CRIM), 550, rue Sherbrooke West, Suite 100, Montreal, Quebec, Canada, H3A 1B9*
**Corresponding author: adenilso@icmc.usp.br*

In this paper, we consider a classical problem of complete test generation for deterministic finite-state machines (FSMs) in a more general setting. The first generalization is that the number of states in implementation FSMs can even be smaller than that of the specification FSM. Previous work deals only with the case when the implementation FSMs are allowed to have the same number of states as the specification FSM. This generalization provides more options to the test designer: when traditional methods trigger a test explosion for large specification machines, tests with a lower, but yet guaranteed, fault coverage can still be generated. The second generalization is that tests can be generated starting with a user-defined test suite, by incrementally extending it until the desired fault coverage is achieved. Solving the generalized test derivation problem, we formulate sufficient conditions for test suite completeness weaker than the existing ones and use them to elaborate an algorithm that can be used both for extending user-defined test suites to achieve the desired fault coverage and for test generation. We present the experimental results that indicate that the proposed algorithm allows obtaining a trade-off between the length and fault coverage of test suites.

Keywords: software testing; finite-state machines; test generation

Received 3 February 2009; revised 28 May 2009
Handling editor: Iain Stewart

1. INTRODUCTION

The problem of generating tests with guaranteed fault coverage, called n-complete tests, for a specification FSM with n states, aka checking experiments and checking sequences, has traditionally been investigated only for the fault domain containing all implementation FSMs with at most n states or even higher; see, e.g. [1–6]. An n-complete test suite guarantees to the test designer exhaustive fault coverage with respect to the given upper bound n on the number of states in implementation machines [7]. The length of n-complete tests is proportional to n^3 [2]; thus their size can become unacceptably large for complex specifications. The test designer may resort to less exhaustive coverage criteria used in FSM-based testing such as state, transition and path coverage; see, e.g. [8, 9]. Indeed, tests that satisfy these criteria scale much better that n-complete tests, but they offer no guaranteed fault coverage in terms of the number of states in faulty implementation FSMs.

We believe that the test designer may want to be able to generate tests while retaining a (reduced) guaranteed fault coverage similar to that offered by n-complete tests. More specifically, the question is how can one generate a p-complete test suite for $p < n$. A solution to this problem would provide control of the degree of test exhaustiveness by varying a the maximal number of states p of faulty state machines whose detection by a p-complete test suite is guaranteed. Methods for building tests providing fault coverage with respect to a number of states in implementation FSMs smaller than that of a specification FSM are thus needed to offer to the test designer a possibility for finding a desirable compromise between the fault coverage and the size of a test suite. Clearly, an n-complete test suite is also p-complete for any $p \leq n$; however, it may well be redundant when $p < n$. Intuitively, the smaller the state number bound for which the fault coverage is guaranteed, the shorter the required tests. We are not aware of any work addressing complete test generation for the case when faulty FSMs do not necessarily have as many states as the specification FSM.

In this paper, we consider a problem of test generation in a more general setting, namely, how a user-defined test suite for a

given deterministic FSM with n states which may contain just an empty sequence can be extended until it becomes p-complete, for a given $p \leq n$.

The generalization considering initial user-defined tests has practical motivations. The test designer may start test generation using approaches based on specification coverage criteria [8], use-cases [10] or test purposes [11]. Test generation can then be completed with additional tests to achieve a required level of fault coverage. A naïve approach would just ignore the existing tests and use a generation method that provides the required fault coverage. This approach likely results in redundant tests. However, if test generation starts with a given test suite, as proposed in this paper, both, specification and fault coverage-driven approaches can in fact be employed together, i.e. it is possible to construct tests that satisfy specification as well as fault coverage criteria.

Solving the generalized test derivation problem, we present sufficient conditions for test suite completeness that are weaker than the ones known in the literature. Based on these conditions, we propose an algorithm that generates a p-complete test suite starting with user-defined initial tests if they are available. The algorithm is also able to determine whether the user-defined test suite satisfies the sufficient conditions; thus, can also be used for test analysis.

We present the results of an experiment that indicate that p-complete test suites, when $p < n$, are indeed shorter than n-complete ones. The results also suggest that p-complete tests suites, for reasonably big p, have a high fault coverage even compared to n-complete test suites.

This paper is organized as follows. In Section 2 we present the necessary basic definitions. In Section 3 we define p-completeness of test suites and discuss tests convergence and divergence in a set of FSMs. These relationships are the basis for defining sufficient conditions for p-completeness in Section 4. In Section 5 an algorithm for generating p-complete test suites is elaborated and its complexity is analysed. We illustrate the algorithm in various scenarios of usage in Section 6. In Section 7 we present the experimental results. In Section 8 we summarize the contributions and discuss the related work. Finally, in Section 9 we present concluding remarks and point to future work.

2. DEFINITIONS

A finite-state machine (FSM) is a deterministic Mealy machine, which can be defined as follows.

DEFINITION 1. *An* FSM M *is a 7-tuple* $(S, s_0, I, O, D, \delta, \lambda)$, *where* S *is a finite set of states with the initial state* s_0, I *is a finite set of inputs,* O *is a finite set of outputs,* $D \subseteq S \times I$ *is a specification domain,* $\delta : D \to S$ *is a transition function and* $\lambda : D \to O$ *is an output function.*

If $D = S \times I$, then M is a *complete* FSM; otherwise, it is a *partial* FSM. As M is deterministic, a tuple $(s, x) \in D$ determines uniquely a defined *transition* of M. For simplicity we use (s, x) to denote the transition, thus omitting its output and final state. A string $\alpha = x_1 \ldots x_k, \alpha \in I^*$, is said to be a *defined* input sequence at state $s \in S$ if there exist $s_1, \ldots, s_{k+1}$, where $s_1 = s$ such that $(s_i, x_i) \in D$ and $\delta(s_i, x_i) = s_{i+1}$, for all $1 \leq i \leq k$. We use $\Omega(s)$ to denote the set of all defined input sequences for state s and Ω_M as a shorthand for $\Omega(s_0)$, i.e. for the input sequences defined for the initial state of M and, hence, for M itself. Figure 1 presents an example of a complete FSM. The initial state is highlighted in bold. The input symbols are a and b and the output symbols are 0 and 1. The label 'x/y' of an edge (transition) from state s to state s' indicates that $\delta(s, x) = s'$ and $\lambda(s, x) = y$, i.e. when the machine is in state s, it responds to input x by producing output y and moving to state s'.

We extend the transition and output functions from input symbols to defined input sequences, including the empty sequence ε, as usual, assuming $\delta(s, \varepsilon) = s$ and $\lambda(s, \varepsilon) = \varepsilon$ for $s \in S$. An FSM M is said to be *initially connected*, if for each state $s \in S$, there exists an input sequence $\alpha \in \Omega_M$, such that $\delta(s_0, \alpha) = s$, called a *transfer* sequence for state s. In this paper, only initially connected machines are considered, since any state that is not reachable from the initial state can be removed without changing the machine's behaviour. A set $C \subseteq \Omega_M$ is a *state cover* for an FSM M if, for each state $s \in S$, there exists $\alpha \in C$ such that $\delta(s_0, \alpha) = s$. A state cover is *minimal* if it contains exactly one transfer sequence for each state. The set $C \subseteq \Omega_M$ *covers* a transition (s, x) if there exists $\alpha \in C$ such that $\delta(s_0, \alpha) = s$ and $\alpha x \in C$. The set C is a *transition cover* (for M) if it covers every defined transition of M. A set of sequences is *initialized* if it contains the empty sequence.

Given a set $C \subseteq \Omega(s) \cap \Omega(s')$, states s and s' are C-*equivalent* if $\lambda(s, \gamma) = \lambda(s', \gamma)$ for all $\gamma \in C$. Otherwise, i.e. if there exists $\gamma \in C$ such that $\lambda(s, \gamma) \neq \lambda(s', \gamma)$, states s and s' are C-*distinguishable*. We say that γ distinguishes s and s' if s and s' are $\{\gamma\}$-distinguishable. States s and s' are *equivalent* if they are $(\Omega(s) \cap \Omega(s'))$-equivalent. Similarly, they are *distinguishable* if they are $(\Omega(s) \cap \Omega(s'))$-distinguishable. We define distinguishability and equivalence of machines as a corresponding relation between their initial states. An FSM

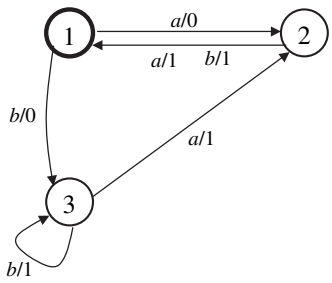


FIGURE 1. A complete FSM M_1.

is *reduced* if all its states are pairwise distinguishable. For instance, the FSM M_1 in Fig. 1 is reduced, since states 1 and 2 are $\{a\}$-distinguishable, states 1 and 3 are $\{b\}$-distinguishable, while states 2 and 3 are $\{aa\}$-distinguishable. In this paper, all the FSMs are assumed to be reduced.[1]

Given sequences $\alpha, \beta, \gamma \in I^*$, if $\beta = \alpha\gamma$, then α is a *prefix* of β, denoted by $\alpha \leq \beta$, and γ is a *suffix* of β. We also say that a prefix of γ *extends* α (*in* β) and that β is an *extension* of α. We denote by pref(β) the set of prefixes of β, i.e. pref$(\beta) = \{\alpha | \alpha \leq \beta\}$. For a set of sequences A, pref(A) is the union of pref(β) for all $\beta \in A$. If $A =$ pref(A), then we say that A is *prefix-closed*. Given a sequence α and $k \geq 0$, we define α^k recursively as follows: $\alpha^0 = \varepsilon; \alpha^k = \alpha\alpha^{k-1}$, if $k > 0$. The *common extensions* of two sequences are the sequences obtained by appending a common sequence to them.

3. TEST PROPERTIES

In this section, we discuss various properties of FSM tests used to formulate a test generation method. First, we formalize the notion of test suite completeness with respect to a given fault domain.

Throughout this paper, we assume that $M = (S, s_0, I, O, D, \delta, \lambda)$ and $N = (Q, q_0, I, O', D', \Delta, \Lambda)$ are a specification FSM and an implementation FSM, respectively. Moreover, n is the number of states of M. We denote by $\Im$ the set of all deterministic FSMs with the same input alphabet as M for which all sequences in Ω_M are defined, i.e. for each $N \in \Im$ it holds that $\Omega_M \subseteq \Omega_N$. The set $\Im$ is called a *fault domain* for M. Given $p \leq n$, let $\Im_p$ be the FSMs of $\Im$ with at most p states, i.e. the set $\Im_p$ is the fault domain for M which represents all faults that can occur in an implementation of M with no more than p states. Faults can be detected by tests, which are input sequences defined in the specification FSM M.

DEFINITION 2. *A defined input sequence of FSM M is called a* test case (*or simply a* test) *of M. A* test suite *of M is a finite prefix-closed set of tests of M. A given test suite T of FSM M is p-complete, $p \leq n$, if for each FSM, $N \in \Im_p$, distinguishable from M, there exists a test in T that distinguishes them.*

Since the distinguishability of FSMs is defined as the corresponding relation of their initial states, tests are assumed to be applied in the initial state. Similarly, FSMs are T-equivalent, for a test suite T, if their initial states are T-equivalent. A *trivial* test suite contains only the empty sequence.

The p-completeness of a test suite provides full *fault coverage* for the fault domain defined by the input alphabet of the specification FSM and maximal number of states p.

The rest of the paper is devoted to the problem of extending a given test suite until it becomes p-complete for a given $p \leq n$. The approach developed in this paper is based on the intricate properties of FSM tests, namely their convergence and divergence. Two defined input sequences of an FSM converge if when applied to the initial state they take the FSM into the same state. Similarly, defined input sequences diverge if they take the FSM from the initial state to different states. We generalize these notions to sets of tests and sets of FSMs. Given a non-empty set of FSMs $\Sigma \subseteq \Im$ and two tests $\alpha, \beta \in \Omega_M$, we say that α and β are Σ-*convergent* if they converge in each FSM of the set Σ. Similarly, we say that α and β are Σ-*divergent* if they diverge in each FSM of Σ. We slightly abuse the notation and say that two tests are M-convergent (M-divergent) when they are $\{M\}$-convergent ($\{M\}$-divergent). Moreover, when it is clear from the context, we drop the set in which tests are convergent or divergent. A set of tests is convergent (divergent) if each pair of its tests are convergent (divergent).

Test convergence and divergence with respect to a single FSM are complementary, i.e. any two tests are either convergent or divergent. However, when a set of FSMs Σ is considered, some tests are neither Σ-convergent nor Σ-divergent. Note that the Σ-convergence relation is reflexive, symmetric and transitive, i.e. it is an equivalence relation over the set of tests. On the other hand, the Σ-divergence relation is irreflexive and symmetric. Consider the FSMs M_1 and M_2 in Figs 1 and 2, respectively. The tests *aaa* and *ba* are $\{M_1, M_2\}$-convergent, whereas the tests *bb* and *ab* are $\{M_1, M_2\}$-divergent. On the other hand, tests *ab* and *baa* are neither $\{M_1, M_2\}$-convergent nor $\{M_1, M_2\}$-divergent since they are M_1-convergent and M_2-divergent.

Several properties of test convergence and divergence can be established.

LEMMA 1. *Given a non-empty set Σ of deterministic reduced FSMs with the same input alphabet, the following properties hold:*

(i) *Common extensions of Σ-convergent tests are also Σ-convergent.*
(ii) *Tests that have Σ-divergent common extensions are also Σ-divergent.*
(iii) *Given two Σ-divergent tests, any test Σ-convergent with one of them is Σ-divergent with the other.*

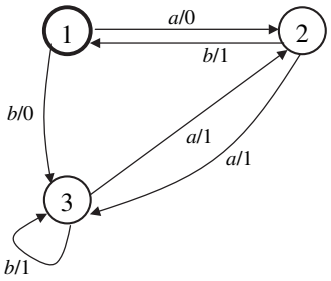


FIGURE 2. A complete FSM M_2.

[1] Test generation considering only reduced state machines is in fact the mainstream in FSM-based testing research; removing this assumption is left for future work.

(iv) *If tests α and $\alpha\varphi^k$ are Σ-divergent, for $k > 1$, then α and $\alpha\varphi$ are also Σ-divergent.*
(v) *If tests α and $\alpha\beta\gamma$ are Σ-convergent and tests α and $\alpha\gamma$ are Σ-divergent, then α and $\alpha\beta$ are also Σ-divergent.*
(vi) *If tests α and $\alpha\gamma$ are Σ-convergent and tests β and $\beta\gamma$ are Σ-divergent, then α and β are also Σ-divergent.*

Proof. Properties (i) and (ii) follow directly from the determinism of the FSMs in Σ, whereas property (iii) comes from the fact that convergence is transitive and divergence is irreflexive.

For property (iv), note that if α and $\alpha\varphi$ converge in some FSM of Σ, then so do α and $\alpha\varphi^2$, by (i) and the transitiveness of convergence. By the same token, α and $\alpha\varphi^3, \alpha\varphi^4, \ldots, \alpha\varphi^k$ would converge, which is a contradiction.

For property (v), suppose α and $\alpha\beta$ converge in some FSM of Σ. Then, due to Lemma 1(ii), $\alpha\gamma$ and $\alpha\beta\gamma$ would also be convergent. Thus, α and $\alpha\gamma$ would converge due to the transitiveness of convergence, which is a contradiction.

For property (vi), suppose that α and β converge in some FSM of Σ. Then, by Lemma 1(i), $\alpha\gamma$ and $\beta\gamma$ would converge. Consequently, β and $\beta\gamma$ would also converge, which is a contradiction. □

Two tests α and β in a given test suite T are *T-separated* if there exist common extensions $\alpha\gamma, \beta\gamma \in T$, such that $\lambda(\delta(s_0, \alpha), \gamma) \neq \lambda(\delta(s_0, \beta), \gamma)$. An important property of T-separated tests is that they are divergent in all FSMs that are T-equivalent to M. Given a test suite T, let $\Im(T)$ be the set of all $N \in \Im$, such that N and M are T-equivalent.

LEMMA 2. *Given a test suite T of an FSM M, T-separated tests are $\Im(T)$-divergent.*

Proof. Let tests α and β be T-separated. Thus, there exist common extensions $\alpha\gamma, \beta\gamma \in T$ and $\lambda(\delta(s_0, \alpha), \gamma) \neq \lambda(\delta(s_0, \beta), \gamma)$. Let N be an FSM T-equivalent to M; thus, we have that $\lambda(\delta(s_0, \alpha), \gamma) = \Lambda(\Delta(q_0, \alpha), \gamma)$ and $\lambda(\delta(s_0, \beta), \gamma) = \Lambda(\Delta(q_0, \beta), \gamma)$. It follows that $\Lambda(\Delta(q_0, \alpha), \gamma) \neq \Lambda(\Delta(q_0, \beta), \gamma)$. Thus, $\Delta(q_0, \alpha) \neq \Delta(q_0, \beta)$, i.e. α and β are N-divergent. □

Consider the FSM M_1 in Fig. 1 and the test suite $T = \text{pref}(\{aaa, baa\})$. We have that tests aa and ba are T-separated since they are extended by a, which distinguishes states $1 = \delta(1, aa)$ and $2 = \delta(1, ba)$; thus, they are $\Im(T)$-divergent. In fact, no deterministic machine that responds to the test suite T as M_1 can reach the same state after the two tests aa and ba.

We now address the problem of demonstrating that tests are $\Im(T)$-convergent, which is more involved than ensuring divergence. Divergence of two tests can be witnessed by different outputs produced by the tests, which are thus divergent in any FSM T-equivalent to M, while convergence of two tests cannot be directly ascertained. However, it can be shown that if a maximal number of states of FSMs in the fault domain is known, and the two tests are $\Im(T)$-divergent with tests reaching all but one state of the FSM M, these two tests must also converge in a same state in any FSM in the fault domain that is T-equivalent to M. Given a test suite T, let $\Im_n(T) = \Im_n \cap \Im(T)$, i.e. the set of FSMs in $\Im$ which are T-equivalent to M and have at most n states. Below we consider only $\Im_n(T)$-convergence, instead of $\Im(T)$-convergence. In particular, we show how the $\Im_n(T)$-convergence of tests can be established based on the existence of an $\Im_n(T)$-divergent set with n tests. Note that, while $\Im(T)$-divergent tests are also $\Im_n(T)$-divergent, the converse does not hold, i.e. there are $\Im_n(T)$-divergent tests that are not $\Im(T)$-divergent. For instance, Lemma 1 can be used to establish the $\Im_n(T)$-divergence of tests from the $\Im_n(T)$-divergence and $\Im_n(T)$-convergence of other tests, but cannot determine their $\Im(T)$-divergence, which requires that the tests in question are T-separated.

LEMMA 3. *Given a test suite T and $\alpha \in T$, let K be an $\Im_n(T)$-divergent set with n tests and $\beta \in K$ be a test M-convergent with α. If α is $\Im_n(T)$-divergent with each test in $K \backslash \{\beta\}$, then α and β are $\Im_n(T)$-convergent.*

Proof. Let $K' = K \backslash \{\beta\}$. The set K' is an $\Im_n(T)$-divergent set and thus it reaches $n - 1$ states of M. As both α and β are $\Im_n(T)$-divergent with each test in K', in any FSM of $\Im_n(T)$, both α and β reach a state that is not reached by the tests in K'. As K' reaches $n - 1$ states and any FSM in $\Im_n(T)$ has at most n states, α and β must reach the same state, i.e. they are $\Im_n(T)$-convergent. □

Consider the FSM M_1 in Fig. 1 and the test suite $T = \text{pref}(\{aaa, baa\})$. We have that the tests ε and aa are $\Im_n(T)$-convergent, since the set $\{\varepsilon, a, b\}$ is $\Im_n(T)$-divergent and the test aa is $\Im_n(T)$-divergent with a and b.

In the next section, we use test divergence and convergence properties to formulate conditions that ensure p-completeness of test suites.

4. SUFFICIENT CONDITIONS FOR p-COMPLETENESS

In this section, we present sufficient conditions for test completeness with respect to the fault domain $\Im_p$, where each FSM has at most p states. These conditions are used to elaborate a generation method in the next section.

The conditions for p-completeness of a test suite T can be divided into two cases, depending on whether $p < n$ or $p = n$. If $p < n$, then it is sufficient show that no FSM in $\Im_p$ is T-equivalent to M, i.e. that $\Im_p(T)$ is empty. However, if $p = n$, then $M \in \Im_n$ and, thus, $\Im_n(T)$ is by definition not empty. To formulate the conditions for dealing with the case of $p = n$, we introduce the notion of convergence-preserving set, for which the M-convergence implies the $\Im_n(T)$-convergence.

DEFINITION 3. *Given a test suite T of an FSM M, a set of tests is $\Im_n(T)$-convergence-preserving (or, simply,*

convergence-preserving) *if all its M-convergent tests are* $\Im_n(T)$*-convergent.*

Note that any M-divergent set is, by definition, convergence-preserving. Consider the FSM M_1 in Fig. 1 and the test suite $T = \text{pref}(\{aaa, baa\})$. The set $\{\varepsilon, a, b\}$ is $\Im_n(T)$-convergence-preserving since it does not contain any M-convergent tests. The set $\{\varepsilon, a, aa, b\}$ is also $\Im_n(T)$-convergence-preserving since the M-convergent tests ε and aa are $\Im_m(T)$-convergent. However, the set $\{\varepsilon, a, b, ba\}$ is not $\Im_n(T)$-convergence-preserving since the tests a and ba are M-convergent but not $\Im_n(T)$-convergent.

THEOREM 1. *Let T be a test suite for an FSM M with n states and* $p \leq n$*. We have that T is a p-complete test suite for M if:*

(i) $p < n$ *and T contains a* $\Im(T)$*-divergent set with* $p + 1$ *tests; or*
(ii) $p = n$ *and T contains an* $\Im_n(T)$*-convergence-preserving initialized transition cover for M.*

Proof. (i) If T contains an $\Im(T)$-divergent set with $p + 1$ tests, then any FSM T-equivalent to M has at least $p + 1$ states. As there exists no such FSM in $\Im_p(T)$, it follows that the test suite T is p-complete.

(ii) Assume now that T contains an $\Im_n(T)$-convergence-preserving initialized transition cover K for M and $p = n$. We prove by contradiction that T is n-complete. Suppose that T is not n-complete. Thus, there exists an FSM $N \in \Im_n(T)$ distinguishable from M. Let φx be a shortest input sequence distinguishing N and M, where x is an input symbol, and hence $\lambda(\delta(s_0, \varphi), x) \neq \Lambda(\Delta(q_0, \varphi), x)$. We show, by induction on the length of φ, that there exists a test in K that is N-convergent with φ. In the base case, we have that φ is the empty sequence. As K includes this sequence, the result follows. The inductive hypothesis is that $\varphi = \beta y$, for some input sequence β and input symbol y, such that β is N-convergent with some test π in K. Since K is a transition cover, it follows that there exists a test υ in K such that υ and π are M-convergent and $\upsilon y \in K$. As K is $\Im_n(T)$-convergence-preserving and $\upsilon, \pi \in K$, it follows that υ and π are $\Im_n(T)$-convergent. As $N \in \Im_n(T)$, we have that υ and π are N-convergent, thus so are υ and β. By Lemma 1(i), we have that υy and by are also N-convergent, and the result follows.

Let χ be a test in K that is N-convergent with φ. As K is a transition cover for M, there exists $\alpha \in K$ such that α and χ are M-convergent and $\alpha x \in K$. As K is $\Im_n(T)$-convergence-preserving, α and χ are $\Im_n(T)$-convergent; hence α and χ are N-convergent since $N \in \Im_n(T)$. It follows that $\lambda(\delta(s_0, \alpha), x) = \lambda(\delta(s_0, \chi), x) = \lambda(\delta(s_0, \varphi), x) \neq \Lambda(\Delta(q_0, \varphi), x) = \Lambda(\Delta(q_0, \chi), x) = \Lambda(\Delta(q_0, \alpha), x)$, i.e. $\lambda(s_0, \alpha x) \neq \Lambda(q_0, \alpha x)$. Thus, αx distinguishes M and N, and, as $\alpha x \in K \subseteq T$, we can conclude that M and N are T-distinguishable, which is a contradiction. Thus, T is n-complete. □

In the next section, the proposed conditions are used to elaborate an algorithm for extending a given (possibly trivial) test suite until it becomes p-complete.

5. ALGORITHM FOR GENERATING p-COMPLETE TEST SUITES

In this section, we present an algorithm for generating p-complete test suites based on Theorem 1 and Lemmas 1–3. Before we introduce the algorithm in Fig. 3, we provide the intuition behind its main steps. Given an FSM M, a (possibly trivial) test suite T and a $p \leq n$, the algorithm generates a test suite that contains T and satisfies the conditions of Theorem 1, and thus the resulting test suite is p-complete. The tests in T are analysed, so that more tests are added only if needed. Depending on the value of p, it is sufficient to do so until the test suite has either an $\Im(T)$-divergent set with $p + 1$ tests or an $\Im_n(T)$-convergence-preserving initialized transition cover.

Note that an $\Im(T)$-divergent set corresponds to a clique in a graph which represents the $\Im(T)$-divergence relation. A *divergence* graph on the tests in T is a graph such that two tests $\alpha, \beta \in T$ are adjacent if α and β are $\Im(T)$-divergent. Thus, an $\Im(T)$-divergent set corresponds to a clique in a divergence graph. If $p < n$, to obtain a p-complete test suite, it is sufficient to guarantee that the corresponding divergence graph contains a clique of size $p + 1$. If $p = n$, however, another approach should be considered, since there exists no $\Im(T)$-divergent set with more than n tests. In this case it is required to ensure the existence of an initialized transition cover that is $\Im_n(T)$-convergence-preserving. Recall that convergence of some tests is implied by divergence and/or convergence of other tests, according to Lemma 1. Thus, the $\Im_n(T)$-convergence and $\Im_n(T)$-divergence relations should be determined incrementally. To this end, we define two relations C and D to represent, respectively, the subsets of $\Im_n(T)$-convergence and $\Im_n(T)$-divergence relationships which are already identified. Initially, the relation C is the identity relation, representing the fact that initially no $\Im_n(T)$-convergence relationships are known, except for the trivial reflexive relationships. On the other hand, the relation D is initially the set of all pairs of T-separated tests according to Lemma 2. These relations are iteratively updated by applying a set of rules that infer new relationships from existing relationships, following Lemma 1. The rules are event-driven, in the sense that they are applied when some relationship is added to C or D. More than one rule can be applicable at the same time.

We derive these rules from Lemma 1 as follows.

Rule 1: If (α, β) is added to C, for each $(\alpha, \chi) \in C$, add (β, χ) to C (transitiveness).
Rule 2: If (α, β) is added to C, then, for all their common extensions $\alpha\varphi, \beta\varphi \in T$, add $(\alpha\varphi, \beta\varphi)$ to C (Lemma 1(i)).

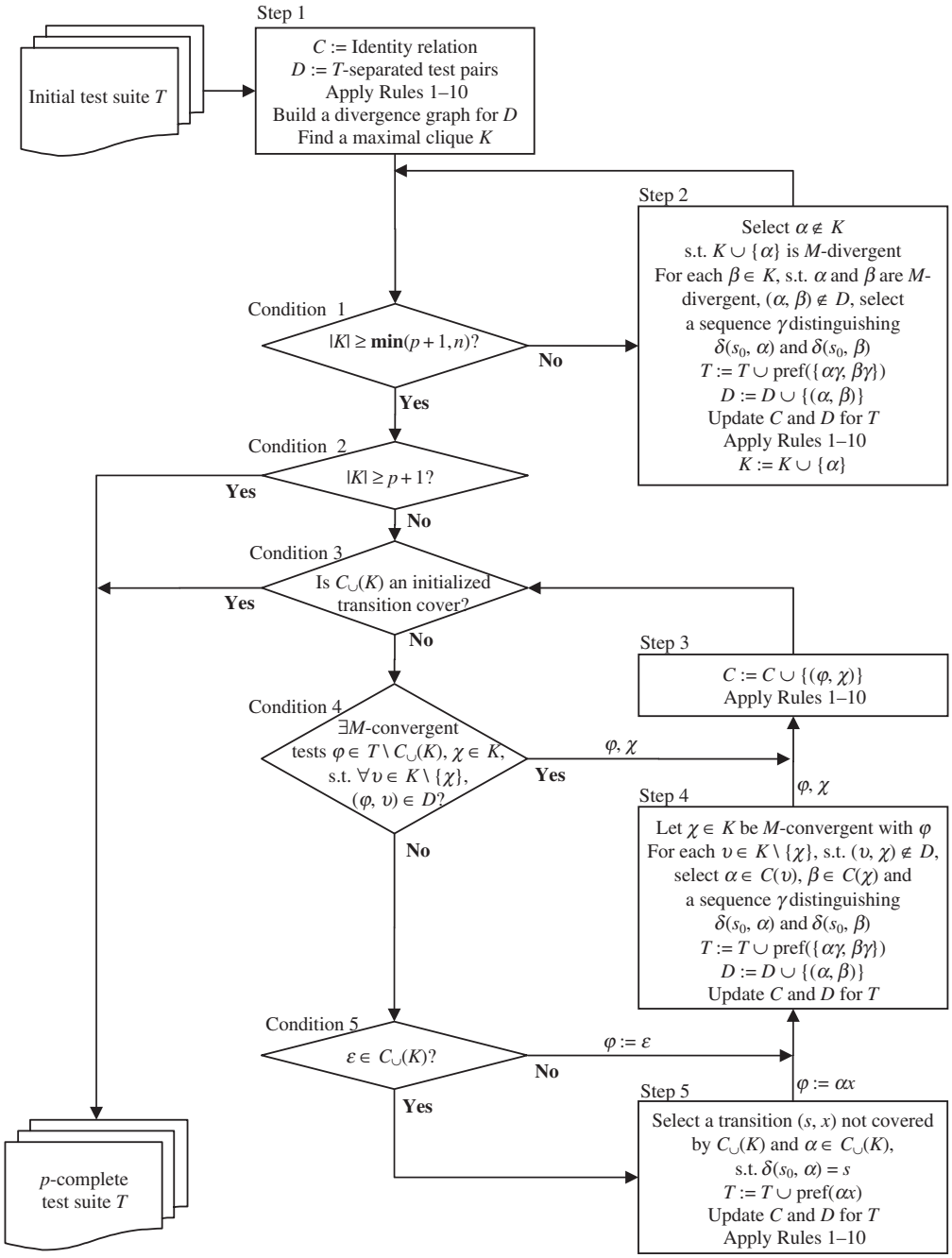


FIGURE 3. Algorithm for generating a p-complete test suite.

Rule 3: If (α, β) is added to D, and they are common extensions of tests α' and β', then add (α', β') to D (Lemma 1(ii)).

Rule 4: If (α, β) is added to C, then, for each $\chi \in T$ if $(\alpha, \chi) \in D$, add (β, χ) to D; if $(\beta, \chi) \in D$, add (α, χ) to D (Lemma 1(iii)).

Rule 5: If (α, β) is added to D, then, for each $\chi \in T$ if $(\alpha, \chi) \in C$, add (β, χ) to D; if $(\beta, \chi) \in C$, add (α, χ) to D (Lemma 1(iii)).

Rule 6: If (α, β), with $\alpha \leq \beta$, is added to D and there exists sequence φ and $k > 1$, such that $\beta = \alpha\varphi^k$, then add $(\alpha, \alpha\varphi)$ to D (Lemma 1(iv)).

Rule 7: If $(\alpha, \alpha\beta\gamma)$ is added to C, and $(\alpha, \alpha\gamma) \in D$, then add $(\alpha, \alpha\beta)$ to D (Lemma 1(v)).

Rule 8: If $(\alpha, \alpha\gamma)$ is added to D, then, for each sequence β such that $(\alpha, \alpha\beta\gamma) \in C$, add $(\alpha, \alpha\beta)$ to D (Lemma 1(v)).

Rule 9: If $(\alpha, \alpha\gamma)$ is added to C, then, for each sequence β such that $(\beta, \beta\gamma) \in D$, add (α, β) to D (Lemma 1(vi)).

Rule 10: If $(\beta, \beta\gamma)$ is added to D, then, for each sequence α such that $(\alpha, \alpha\gamma) \in C$, add (α, β) to D (Lemma 1(vi)).

If, to achieve p-completeness, more tests are added to T, the relations C and D should also be extended with the new tests. Recall that a test suite is prefix-closed. Thus, adding a test α to T results in the addition of all prefixes of α. If α is added to T, the identity pair (α, α) has to be added to C. Moreover, the test pairs that are T-separated in the extended test suite must be added to D.

Initially, the algorithm finds a largest $\Im(T)$-divergent set, which corresponds to a clique in the divergence graph. If the determined clique has more than $p+1$ tests, then the test suite is already p-complete. Recall, however, that no $\Im(T)$-divergent set has more than n tests. Thus, when the clique has fewer than $\mathbf{min}(p+1, n)$ tests, a test that is not in the clique may be selected to extend it. It is possible to do so if the test to be added is $\Im(T)$-divergent with all tests in the clique. Hence, if the test is not $\Im(T)$-divergent with some test in the clique, it is sufficient to add tests to T, so that the two tests become T-separated. If $p = n$, an n-clique, i.e. a clique with n nodes, can thus be eventually obtained, but this is not sufficient for ensuring the n-completeness, according to Theorem 1. In this case, it is additionally required to ensure that T contains an initialized transition cover that is $\Im_n(T)$-convergence-preserving.

We now show how such a transition cover can be obtained from an n-clique. As the relation C is an equivalence relation, it induces a partition on the tests in T. Given a test $\alpha \in T$, let $C(\alpha) = \{\beta | (\alpha, \beta) \in C\}$ be the block of the partition induced by C that contains α. Let K be an n-clique. We denote by $C_\cup(K)$ the union of the blocks which have a test in K, i.e. $C_\cup(K) = \{\beta | (\alpha, \beta) \in C, \alpha \in K\}$. Recall that in an $\Im_n(T)$-divergent set, no tests are M-convergent, i.e. an $\Im_n(T)$-divergent set is trivially $\Im_n(T)$-convergence-preserving. Thus, the set of tests $C_\cup(K)$ is $\Im_n(T)$-convergence-preserving. To ensure that $C_\cup(K)$ is an initialized transition cover for M, we might need to extend it. We say that a test α is added to $C_\cup(K)$, when (α, β) is added to C, where $\beta \in C_\cup(K)$ is a test $\Im_n(T)$-convergent with α. Lemma 3 indicates that a test can be added to $C_\cup(K)$ if it is $\Im_n(T)$-divergent with $n-1$ tests in K. It is sufficient to show that the test which is not in $C_\cup(K)$ is $\Im_n(T)$-divergent with the $n-1$ tests of the clique. If the tests form a pair in D, then they are already $\Im_n(T)$-divergent. Otherwise, tests could be added so that the two tests become T-separated and, thus, $\Im_n(T)$-divergent. The set $C_\cup(K)$ resulting from the addition of (α, β) to C remains $\Im_n(T)$-convergence-preserving. Therefore, to obtain an n-complete test suite, it is sufficient to add suitable tests to $C_\cup(K)$ until it becomes an initialized transition cover for M.

Depending on the tests that are already in the sets $C_\cup(K)$ and T, there are three cases to consider. The first case occurs when tests can be added to $C_\cup(K)$ without adding tests to T, i.e. when there are tests that already satisfy the condition of Lemma 3. As a result, the number of blocks in the partition induced by C is decreased, since the blocks to which these tests belong are merged in the resulting partition. It is important to note that this case may result in $C_\cup(K) = T$. Thus, if T is also a transition cover for M (recall that T is prefix-closed and, thus, initialized), then T is n-complete.

In the remaining cases, adding tests to $C_\cup(K)$ requires new tests be first added to T, making Lemma 3 applicable. If the empty sequence is not in $C_\cup(K)$, tests are added so that the empty sequence can be added to $C_\cup(K)$. Then, $C_\cup(K)$ becomes initialized. Finally, if there is a transition not covered by $C_\cup(K)$, a test is added to $C_\cup(K)$ so that it becomes covered. Thus, $C_\cup(K)$ eventually becomes a transition cover. As it is also initialized and $\Im_n(T)$-convergence-preserving, by Theorem 1, T is p-complete.

The above discussion leads to the algorithm, presented in Fig. 3, for extending a test suite until its p-completeness can be guaranteed. Labels on edges connecting steps and conditions, φ and χ, denote the tests defined in a precedent step or condition.

We illustrate the algorithm in Section 6. It is important to note that in several steps of the algorithm we do not restrict the selection of tests with the required properties. Various selection strategies can be used there, for instance, the distinguishing sequences selected in Step 4 can be obtained from identification sets obtained *a priori*, as in the methods W [2, 3] and Wp [4], or *on-the-fly*, as in H method [12]. Moreover, the sequences needed to reach a state (in Step 2) or to cover a transition (in Step 5) can be selected using different strategies, such as a breadth-first traversal of the FSM or a transition tour. We believe that several alternative selection strategies should become options in a tool implementing the proposed algorithm for constructing complete tests for FSMs.

In the remainder of this section, we prove that the algorithm terminates and the obtained test suite is indeed p-complete. This discussion is independent from the strategies for sequence selection. Then, we show that the algorithm can be executed in polynomial time if the strategies used to select sequences can be executed in polynomial time.

THEOREM 2. *The algorithm terminates with a p-complete test suite for M.*

Proof. The algorithm contains four cycles. We show that each cycle can be executed a finite number of times and, thus, the algorithm indeed terminates. Then, we prove that the resulting test suite is p-complete.

In the cycle that contains Step 2, the size of the clique is increased in each iteration, until the required size is reached.

Thus, this cycle can only be executed a finite number of times. The other cycles correspond to the three cases discussed above. At the end, in the execution of each cycle, (φ, χ) is added to C and Rules 1–10 are applied as long as possible, in Step 3. The steps that precede Step 3 guarantee that φ and χ are $\Im_n(T)$-convergent. For instance, if necessary, Step 4 adds tests to T so that Lemma 3 can be applied.

Cycle 1 corresponds to the executions where Condition 3 does not hold, but Condition 4 does, i.e. $C_\cup(K)$ is not an initialized transition cover and there exists a test that can be added to $C_\cup(K)$ without adding new tests to T. As the number of tests in $C_\cup(K)$ is increased in each execution of this cycle and the number of tests in T is not changed, after a finite number of executions, the cycle can no longer be executed without involving other cycles. Note that those cycles add new tests to T, possibly increasing the number of blocks. However, as will be shown, those cycles also can be executed a finite number of times and, thus, the number of executions of Cycle 1 is bounded.

Cycle 2 corresponds to the executions where Conditions 3–5 do not hold, i.e. the empty sequence is not in $C_\cup(K)$. Then, the empty sequence is added to $C_\cup(K)$ and, thus, this cycle can be executed at most once.

Cycle 3 corresponds to the executions where Conditions 3 and 4 do not hold, but Condition 5 does, i.e. $C_\cup(K)$ is initialized but is not a transition cover. Step 5 selects a transition (s, x) that is not covered by $C_\cup(K)$ and a test $\alpha \in C_\cup(K)$, $\delta(s_0, \alpha) = s$. The test αx is added to T. Then, Step 4 adds tests to T so that αx is added to $C_\cup(K)$ and, thus, the transition (s, x) becomes covered by $C_\cup(K)$. Therefore, each execution of this cycle requires that a transition not covered by $C_\cup(K)$ exists and it results in the covering of at least one transition. This cycle can thus be executed at most as many times as the number of transitions of M.

Therefore, the algorithm actually terminates since all cycles can be executed only a finite number of times.

We now show that the obtained test suite is p-complete. When the algorithm terminates, either Condition 2 or Condition 3 holds. If Condition 2 holds, the clique has $p + 1$ tests; then the test suite contains an $\Im(T)$-divergent set with $p + 1$ tests and, thus, is p-complete, by Theorem 1. If Condition 3 holds, the set $C_\cup(K)$ is an initialized transition cover for M. As $C_\cup(K)$ is $\Im_n(T)$-convergence-preserving, by Theorem 1, the resulting test suite T is p-complete. □

We next discuss the worst case time complexity of the algorithm and the upper bounds of p-complete test suites. When appropriate, we discriminate the cases $p < n$ and $p = n$, since they have different worst case time complexity. Recall that in several steps of the algorithm we do not restrict the selection of tests with the required properties. However, we show that the algorithm terminates in polynomial time, as long as tests are selected in polynomial time. When appropriate, we indicate strategies for doing so.

Initially, the algorithm needs to find a maximal clique in a graph. This problem is known to be NP-complete and, thus, an optimal solution cannot be found in reasonable time for some instances. Nonetheless, the algorithm does not rely on the fact that the clique found is a largest one. Indeed, if a suboptimal clique is found, it will be extended to the required size by adding new tests to create T-separability relationships omitted when a subclique is chosen. Thus, it is always possible to reduce the time needed to find a largest clique at a price of increasing the test suite. In other words, the NP-completeness of the maximal clique problem does not imply that the proposed algorithm does not scale. For instance, for finding cliques polynomial time greedy-based algorithms can be used; see, [13, 14]; optimization techniques have also been used to solve this problem, which can handle very large graphs in reasonable time [15, 16]. Nevertheless, even in the worst case when the maximal clique found is smaller than a largest one, a complete test suite can be obtained, though its irreducibility might be hard to claim.

In Steps 2 and 5, the algorithm requires that tests are added to obtain divergence relationships. Specifically, given two tests, it is necessary to select a sequence that distinguishes the states reached by them. This problem can be solved by a breadth-first search in a product machine, as defined in [6], in $O(v + w)$, where v and w are the numbers of vertices and edges in the graph, respectively. The product machine has at most n^2 vertices and kn^2 edges, where n is the number of states and k is the number of inputs of M. Thus, the time required to find shortest distinguishing sequences is $O(n^2 + kn^2) = O(kn^2)$ [5].

The algorithm requires the manipulation of the tests in T; thus, its complexity depends on the number of tests included in T at a given stage of the algorithm. As the execution of the algorithm changes this number, we follow a conservative approach, considering the number of tests in the resulting test suite, which is certainly larger than the number of tests actually manipulated by the algorithm at the execution of a given step. Thus, let l be the number of tests in the test suite obtained by the algorithm.

The application of Rules 1–10 in the algorithm is event-driven, in the sense that the rules are applied when new relationships are added to C or D. Thus, they are applied at most once for each pair of tests. Thus, there are $O(l^2)$ pairs of tests. As the relation C is an equivalence relation, it can be represented by the partition it induces. Using a *union-find* algorithm, the time for performing the operations on the partition is $O(\text{Ack}^{-1}(l, l))$, where $\text{Ack}^{-1}(l, l)$ is the inverse of the extremely quickly-growing Ackermann function. For any reasonable value of l, $\text{Ack}^{-1}(l, l)$ is less than five, i.e. the running time of the operations on C is effectively a small constant [17].

As the execution of the algorithm advances, the size of the relation D approximates $l(l - 1)/2$. Thus, we represent this relation in a symmetrical matrix, so that the operations on D can be performed in constant time, at the price of using $O(l^2)$ space.

We now discuss the complexity of executing each rule. Based on the discussion above, we assume that the operation of

verifying whether a pair is in C or in D can be completed in constant time. Thus, we show that all the rules can be executed in $O(l)$ time. Rule 1 enforces the transitiveness of relation C, which, in the worst case, requires analysing all tests. This can be done in $O(l)$. Rule 2 requires the identification of common extensions of two tests, which can be achieved, in the worst case, by inspecting all tests, i.e. in $O(l)$. Similarly, Rule 3 can be executed by checking whether the tests in a pair added to D are common extensions of their prefixes. In the worst case, the time required for Rule 3 is the number of prefixes of the longest test, which is $O(l)$ when the test suite contains a single test. Both Rules 4 and 5 require the inspection of all tests, i.e. in $O(l)$. Rules 6–10 are applicable if one test is a prefix of the other, which can be checked in $O(l)$. For Rules 6 and 7, it is sufficient to check if the suffix is in an appropriate form. Rules 8–10 require inspecting all the tests and, thus, their complexity is $O(l)$.

As there are $O(l^2)$ pairs and each may need operations with $O(l)$ time, the worst case complexity time for applying Rules 1–10 is $O(l^3)$. It is important to note that, although the rules are applied in various steps executed several times, each pair is analysed at most once, when it is added to the respective relation.

The algorithm contains four cycles. In the cycle that contains Step 2, the size of the clique is increased, until the required size is reached. If $p < n$, in the worst case, the cycle can be executed $p+1$ times. In each iteration, Step 2 requires finding at most p distinguishing sequences. Thus, in the worst case, the execution time complexity of this cycle, which is the only cycle executed by the algorithm if $p < n$, is $O(p^2)O(kn^2) = O(kp^2n^2)$. If $p = n$, the cycle can be executed n times, requiring the search for $n-1$ distinguishing sequences; thus, in the worst case, the execution time of this cycle is $O(n^2)O(kn^2) = O(kn^4)$.

The cycle where Condition 4 holds can be executed at most $l-n$ times, i.e. $O(l)$. Condition 4 requires inspecting n tests in K and at most $l-n$ tests in $T\backslash C_{\cup}(K)$, totalling at most $O(nl)$ pairs. For each pair, the other $n-1$ tests in K are analysed. Thus, the execution time of Condition 4 is $O(n^2l)$, and the cycle that contains it requires a time of $O(n^2l^2)$.

The other cycles require the execution of Step 4, which finds $n-1$ distinguishing sequences; thus, its execution time is $O(n)O(kn^2) = O(kn^3)$. This step is involved in two cycles, which can be executed at most as many times as the number of defined transitions, i.e. $O(kn)$. Thus, these cycles are executed in $O(kn)O(kn^3) = O(k^2n^4)$.

The cost of the algorithm is thus $O(\text{CLIQUE}) + O(n^2l^2 + l^3 + k^2n^4)$, where $O(\text{CLIQUE})$ is the time required by the algorithm chosen for finding a maximal clique. Thus, the algorithm runs in polynomial time, after a maximal clique has been found.

The algorithm proposed in this paper can generate p-complete test suites even when $p < n$. The authors are not aware of other methods with such property. It allows the test designer to find a compromise between the cost of complete tests and the size of the fault domain where the completeness is guaranteed. For instance, if n-complete test suites are too expensive to be used, the test designer may choose using, say, $(n/2)$-complete test suites, which nonetheless ensures that if any faulty implementation has at most $n/2$ states, it will be caught by the test suite. Moreover, by increasing the value of p, the test designer can enlarge the tests until an implementation bug is discovered; at the end, there is well-defined guaranteed fault coverage in terms of the number of states.

Finally, we discuss the upper bounds of p-complete test suites when $p < n$. For $p = n$, it is known that the size of an n-complete test suite can reach $O(kn^3)$, for complete FSMs [2] or $O(kn^4)$, for reduced partial FSMs [5]. For $p < n$, in the worst case, a p-complete test suite contains $p+1$ tests, reaching $p+1$ distinct states, and each pair of states requires a distinct distinguishing sequence. Thus, a p-complete test suite needs at most $p(p+1)$ tests. Any state in an initially connected FSM can be reached by a test no longer than $n-1$. In a complete FSM, any pair of states can be distinguished by a sequence of at most $n-1$ inputs. Thus, there is a p-complete test suite for a complete FSM with at most $p(p+1)2(n-1)$ inputs, i.e. $O(p^2n)$. In a partial FSM, any pair of distinguishable states can be distinguished by a sequence no longer than $n(n-1)/2$ [6]. There is thus a p-complete test suite for a partial reduced FSM with at most $p(p+1)(n-1+n(n-1)/2) = p(p+1)((n+2)(n-1)/2)$ inputs, i.e. $O(p^2n^2)$. Therefore, when $p < n$, the upper bounds for p-complete test suites are lower than those for n-complete test suites by a factor of $O((p/n)^2)$. It is important to note that reasonable choices have to be made when sequences are selected for the algorithm to obtain a test suite not exceeding these bounds. For example, a longer test suite would be obtained if distinguishing sequences selected in Step 2 are not the shortest ones (e.g. longer than $n-1$).

In Section 7, we provide the results of experimental evaluation of the length of p-complete test suites.

6. EXAMPLES

In this section we present examples of the execution of the algorithm for the FSM M_1 in Fig. 1. In the first example, the algorithm generates a series of test suites with increasing fault coverage for various values of p. In the second example, the algorithm is given a test suite that already has the desired fault coverage. As expected, the algorithm terminates without adding new tests, even though the test suite does not satisfy the existing sufficient conditions. This demonstrates the fact that the proposed conditions are weaker and the algorithm improves the state-of-the-art in test coverage analysis. Finally, in the last example, we illustrate that the algorithm can be used to extend a given test suite until complete fault coverage is achieved.

6.1. Incremental generation

We consecutively execute the algorithm to obtain p-complete test suites T_p, $p = 1, 2, 3$. Initially a test suite contains only the empty sequence $\{\varepsilon\}$.

Case $p = 1$: As the divergence graph has just one vertex, it has a 1-clique $K = \{\varepsilon\}$. We have that Condition 1 does not hold. Then, in Step 2, a test $\alpha = a$ is added to T, which reaches a new state and we select the sequence $\gamma = a$ to distinguish $\delta(s_0, a)$ and $\delta(s_0, \varepsilon)$. We then add aa to T, resulting in a 2-clique $K = \{\varepsilon, a\}$. The resulting test suite $T_1 = \text{pref}(aa)$ is 1-complete.

Case $p = 2$: We now want to obtain a 2-complete test suite, starting the algorithm with T_1. The algorithm finds the 2-clique $K = \{\varepsilon, a\}$. We have that Condition 1 does not hold. A test $\alpha = b$ reaches a new state is added in Step 2. Then, tests are added to T so that b and ε, as well as b and a, are T-separated. For b and ε, we add ba to T. For b and a we add aaa and baa. Then, the clique is extended with b, resulting in a 3-clique $K = \{\varepsilon, a, b\}$. The final test suite $T_2 = \text{pref}(\{aaa, baa\})$ is 2-complete.

Case $p = 3$: Finally, we execute the algorithm to obtain a 3-complete test suite. The test suite T_2 is used to initialize the algorithm and the clique $K = \{\varepsilon, a, b\}$ is found. We have that Condition 1 holds, but Condition 2 does not. Then, it is necessary to add tests to T to obtain an $\Im_n(T)$-convergence-preserving initialized transition cover. The relation D is represented in the divergence graph in Fig. 4a. We use $\Upsilon(C)$ to represent the partition induced by the relation C. In this case, $\Upsilon(C) = \{\{\varepsilon\}, \{a\}, \{aa\}, \{aaa\}, \{b\}, \{ba\}, \{baa\}\}$.

Condition 3 holds for $\varphi = aa$ and $\chi = \varepsilon$. Then, after execution of Step 3, we obtain $\Upsilon(C) = \{\{\varepsilon, aa\}, \{a, aaa\}, \{b\}, \{ba\}, \{baa\}\}$ and the divergence graph in Fig. 4b. For simplicity, only one test per block is shown in this and the following divergence graphs, since the relationships of the omitted tests can be inferred.

Condition 4 does not hold, but Condition 5 does. Then, we select $\alpha = b$ and $x = a$ and execute Steps 3–5 for $\varphi = ba$ and $\chi = a$. For $\upsilon = \varepsilon$, no additional tests are necessary. However, for $\upsilon = b$, we add the test $baaa$ to T, so that b and ba are T distinguishable. Then, after adding (ba and a) to C and applying Rules 1–10, we obtain $\Upsilon(C) = \{\{\varepsilon, aa, baa\}, \{a, aaa, ba, baaa\}, \{b\}\}$.

As $C_\cup(K)$ is not a transition cover, Step 5 is executed, extending T to cover a yet uncovered transition. We select the transition (2, b), the tests $\alpha = aaa$, $x = b$ and execute Steps 3 and 4 for $\varphi = aaab$ and $\chi = \varepsilon$. For $\upsilon = a$, we add the test $aaaba$ to T. For $\upsilon = b$, it is not necessary to add new tests since b and $aaab$ are already T-separated. The application of Rules 1–10 results in the partition $\Upsilon(C) = \{\{\varepsilon, aa, baa, aaab\}, \{a, aaa, ba, baaa, aaaba\}, \{b\}\}$.

The algorithm continues to cover the transition (3, b). In Step 5, we select the test bb to be added to T. Then, Steps 4 and 5 are executed for $\varphi = bb$ and $\chi = b$. The test $bbaa$ is added to T. The resulting partition is $\Upsilon(C) = \{\{\varepsilon, aa, baa, aaab, bbaa\}, \{a, aaa, ba, baaa, aaaba, bba\}, \{b, bb\}\}$. As $C_\cup(K)$ is an initialized transition cover, the resulting test suite $T = \text{pref}(\{aaaba, baaa, bbaa\})$, which has length 16 and requires three resets, is 3-complete.

The example shows that the algorithm allows generating tests that require fewer resets than the existing methods. In particular, the Wp [4] and H method [12] generate the same test suite $T_{3-\text{complete}} = \text{pref}(\{aaa, aba, baaa, bbaa\})$ of length 18, which requires four resets.

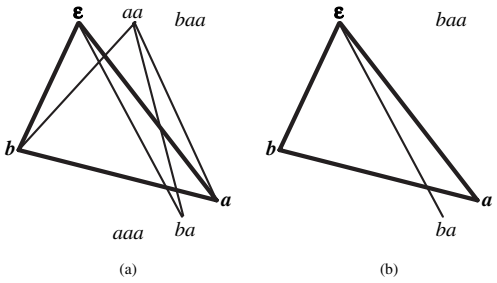


FIGURE 4. Divergence graphs obtained during the generation of $T = \text{pref}(\{aaaba, baaa, bbaa\})$.

6.2. Confirming p-completeness

We illustrate the execution of Algorithm 1 with the FSM M_1 in Fig. 1, initial test suite $T = \text{pref}(\{aaa, abb, baba, bbab\})$ and $p = n = 3$. We show that T is indeed an n-complete test suite for M_1, without adding more tests. Note that the n-completeness of T cannot be established using the conditions proposed in [12, 18] in the sense that this test suite does not satisfy either of the two conditions but is, nevertheless, 3-complete.

Initially, C is the identity relation. After populating D with the T-separated tests (Rules 1–10 are not applicable), we obtain the divergence graph in Fig. 5a. The algorithm then finds the maximal clique $K = \{\varepsilon, b, ba\}$.

As Condition 4 holds for $\chi = \varepsilon$ and $\varphi = ab$, Step 3 is executed, adding ab to $C_\cup(K)$. After applying Rules 1–10, the following relationships are determined:

(i) (b, abb) is added to C (Rule 2);
(ii) (bb, ab) is added to D (Rule 4);
(iii) (b, a) is added to D (Rule 3);
(iv) (abb, ε), (abb, aa), (abb, ab), (abb, bab), (abb, a), (abb, ba) are added to D (Rule 4).

We have that $\Upsilon(C) = \{\{\varepsilon, ab\}, \{b, abb\}, \{ba\}, \{a\}, \{aa\}, \{aaa\}, \{bab\}, \{baba\}, \{bb\}, \{bba\}, \{bbab\}\}$. Figure 5b presents the updated divergence graph. We represent the tests in $C_\cup(K)$ in bold type and the edges added to the graph are dashed.

As Condition 4 holds for $\chi = ba$ and $\varphi = a$, Step 4 is executed. Then, (ba, a) is added to C and Rules 1–10 are applied. The following relationships are determined:

(i) (ab, bab) is added to C (Rule 2);
(ii) (ε, bab) is added to C (Rule 1);

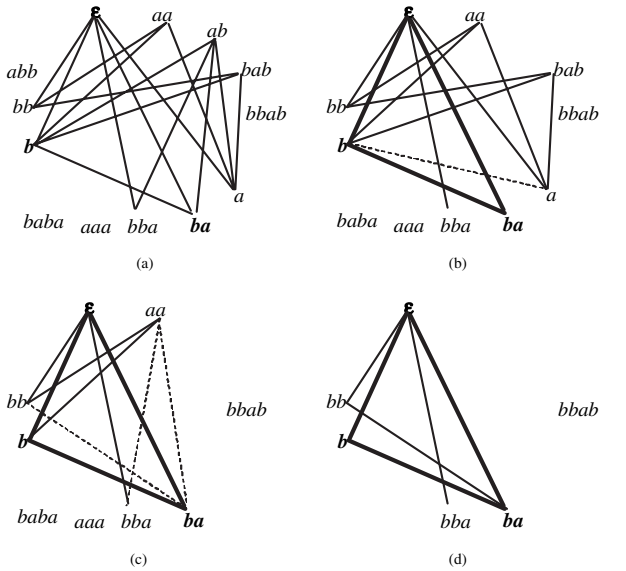


FIGURE 5. Divergence graphs obtained during the execution with $T = \text{pref}(\{aaa, abb, baba, bbab\})$.

(iii) (ba, bab), (ba, aa), (bba, aa), (bab, bba) are added to D (Rule 4);
(iv) (bb, a) is added to D (Rule 3);
(v) (bb, ba) is added to D (Rule 5).

In Fig. 5c, we present the updated divergence graph. We have that $\Upsilon(C) = \{\{\varepsilon, ab, bab\}, \{ba, a\}, \{b, abb\}, \{aa\}, \{aaa\}, \{baba\}, \{bb\}, \{bba\}, \{bbab\}\}$.

Now, Condition 4 holds for $\chi = \varepsilon$ and $\varphi = aa$. The execution of Step 4 adds (ε, aa) to C and Rules 1–10 are applied, resulting in the following additional relationships:

(i) (ab, aa), (bab, aa) are added to C (Rule 1);
(ii) (a, aaa), $(aaa, baba)$ are added to C (Rule 2);
(iii) $(ba, baba)$, $(a, baba)$ are added to C (Rule 1);
(iv) (aaa, bab), (aaa, ab), (aaa, aa), (aaa, ε), (aaa, abb), (aaa, bb), (aaa, b), $(baba, bab)$, $(baba, ab)$, $(baba, aa)$, $(baba, \varepsilon)$, $(baba, abb)$, $(baba, bb)$, $(baba, b)$ are added to D (Rule 4).

In Fig. 5d, we present the updated divergence graph. We have that $\Upsilon(C) = \{\{\varepsilon, aa, ab, bab\}, \{ba, a, aaa, baba\}, \{b, abb\}, \{bb\}, \{bba\}, \{bbab\}\}$.

Condition 4 holds once more, selecting $\chi = b$ and $\varphi = bb$. Then, (b, bb) is added to C and Rules 1–10 are applied. Now, we have that $C_\cup(K)$ is an initialized transition cover for M_1; thus T is indeed 3-complete.

The example demonstrates that the proposed sufficient conditions are weaker than the existing ones, as the latter cannot establish the test suite completeness.

6.3. Completing user-defined test suites

We now illustrate how the algorithm can be used to extend a user-defined test suite, obtaining a p-complete test suite. Consider again the FSM M_1 in Fig. 1 and $p = 1, 2, 3$. In this example, we use a test suite $T_{\text{tour}} = \text{pref}(bbabaa)$, which is derived from a transition tour for M_1. Figure 6a presents the corresponding divergence graphs of T_{tour}. Note that the set $\{\varepsilon, b, bba\}$ is $\Im_n(T)$-divergent. Thus, when the algorithm is executed with T_{tour} and $p = 1$ or $p = 2$, no test is added, since the test suite T already satisfies the conditions for 1- and 2-completeness of Theorem 1.

Let $T = T_{\text{tour}}$. The set $K = \{\varepsilon, b, bba\}$ is the only maximal 3-clique in the divergence graph. Step 5 is executed, selecting the transition $(3, b)$ and $\alpha = b$. Then, Step 4 is executed for $\varphi = bb$ and $\chi = b$. For $\upsilon = \varepsilon$, we add the test a to T. For $\upsilon = bba$, we add $bbba$ to T. After applying Rules 1–10, we obtain the following partition $\Upsilon(C) = \{\{\varepsilon\}, \{b, bb, bbb\}, \{bba, bbba\}, \{a\}, \{bbab\}, \{bbaba\}, \{bbabaa\}\}$. The resulting divergence graph is presented in Fig. 6b.

As $C_\cup(K)$ is initialized but is not a transition cover, Step 5 is executed. We select the transition $(2, b)$ and the test $\alpha = bba$. Then, we execute Steps 3 and 4 for $\varphi = bbab$ and $\chi = \varepsilon$. For $\upsilon = bba$, we may add test $bbaa$, so that bba and $bbab$ become T-separated. However, as $(bba, bbba) \in C$, we instead add $bbbaa$, which ensure the T-separability of $bbbaa$ and $bbab$. After applying Rules 1–10 (specifically, Rule 5), $(bba, bbab)$ is added to D, as required. This choice is motivated by the fact that adding $bbbaa$ instead of $bbaa$ would not require an additional reset. After applying Rules 1–10, we obtain $\Upsilon(C) = \{\{\varepsilon, bbab\}, \{b, bb, bbb\}, \{bba, bbba\}, \{a, bbaba\}, \{bbabaa\}, \{bbbaa\}\}$. Figure 6c presents the resulting graph.

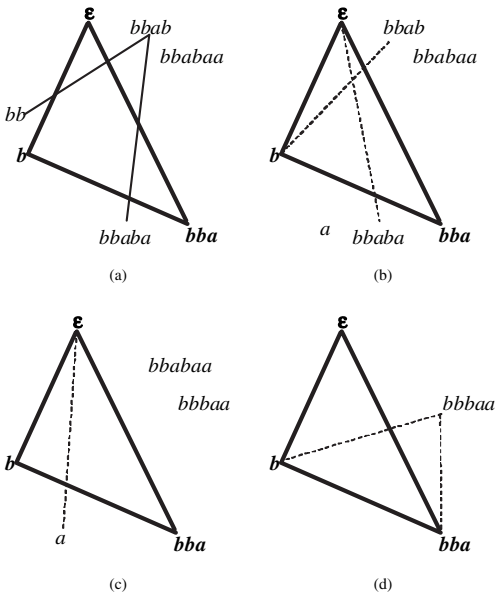


FIGURE 6. Divergence graphs obtained during the generation of $T = \text{pref}(\{a, bbabaab, bbbaa\})$.

As $C_{\cup}(K)$ is not a transition cover, we execute Step 5 for the transition $(1, a)$ and $\alpha = bbab$. Steps 3 and 4 are executed for $\varphi = bbaba$ and $\chi = bba$. For $\upsilon = \varepsilon$, it is not necessary to add tests to T. For $\upsilon = b$, we add $bbabaab$ to T. After applying Rules 1–10, we obtain $\Upsilon(C) = \{\{\varepsilon, bbab\}, \{b, bb, bbb\}, \{bba, a, bbba, bbaba\}, \{bbbaa, bbabaa\}\}$. Figure 6d presents the resulting graph.

We have that, with $\varphi = bbbaa$ and $\chi = \varepsilon, (\varphi, \chi)$ can be added to C without adding new tests. Now we have that $C_{\cup}(K) = T = \text{pref}(\{a, bbabaab, bbbaa\})$ is a transition cover, and thus 3-complete, whose length is 16. The example shows how a user-defined test suite (derived using a specification coverage criterion in this particular example) can be extended using the proposed algorithm to achieve the desired fault coverage.

Finally, we compare the proposed algorithm with a simplistic approach for extending a user-defined test suite to ensure its p-completeness. Instead of analysing the tests furnished by the user, a p-complete test suite is otherwise generated and added to the used-defined test suite. If $p < n$, a p-complete tests suite can obtained as follows. A (minimal) set with $p + 1$ tests, reaching $p + 1$ states in M_1, can be determined. Then, to each pair of tests, a sequence that distinguishes the reached states is appended. The obtained test suite, which is p-complete, is added to T_{tour}, thus ignoring the tests already in T_{tour}. For instance, a 2-complete test suite for M_1 obtained in this way is $T_{2-\text{complete}} = \text{pref}(\{aaa, baa\})$ of length 8. However, recall that the user-defined test suite T_{tour}, which has length 7, is already 2-complete and no additional test is added by the proposed algorithm. Nevertheless, the simplistic approach would just add $T_{2-\text{complete}}$ to T_{tour}, resulting in a test suite of length 15. Additionally, note that the test suite $T_{2-\text{complete}}$ corresponds to the 2-complete test suite obtained by the proposed algorithm in Section 6.1, where the test suite was incrementally generated from a trivial test suite. Similarly, if $p = n$, a 3-complete test suite can be generated by an existing method and added to T_{tour}. For instance, as mentioned before, the Wp and H methods generate the test suite $T_{3-\text{complete}} = \text{pref}(\{aaa, aba, baaa, bbaa\})$. Therefore, the resulting test suite would be $T_{\text{tour}} \cup T_{3-complete}$, whose length is 25, while the length of the test suite produced by the proposed method is 16.

7. EXPERIMENTAL RESULTS

In this section we present the experimental results on the fault coverage of p-complete test suites. We also show how the length of a p-complete test suite grows as the value of p increases. The average length of the p-complete test suites is compared with the upper bound discussed in Section 5.

Although experiments involving 'realistic' FSM designed by human testers are highly desirable, the manual generation of a sufficient number of FSMs could be excessively expensive. Thus, in the experiments, we used randomly generated FSMs, as it is usual in experimental evaluation of FSM-based test generation methods [9, 19, 20].

Complete reduced FSMs are generated as follows. Initially, sets of states, inputs and outputs with the required number of elements are generated. The generation then proceeds in two phases. In the first phase, a state is selected as the initial state and marked as 'reached'. Then, for each state s not marked as 'reached', we select a reached state s', an input x and an output y and add to the machine being generated a transition from s' to s with input x and output y, and mark s as 'reached'. When this phase is completed, an initially connected FSM is obtained. In the second phase, transitions are added to the machine by randomly selecting two states, an input and an output, until it is complete. We then check if the FSM is reduced. A non-reduced FSM is discarded and another FSM is generated.

In the experiments, we randomly generated 100 complete reduced FSMs with 50 states, four inputs and four outputs. For each FSM M, we incrementally generated p-complete test suites,[2] $p < n = 50$. The average length of the obtained test suites is shown in Fig. 7. Recall that the upper bound for p-complete test suites is $O(p^2n)$. We note that, although the length of p-complete test suites grows nonlinearly, it is well below the theoretical limit (the curve p^2 is also included in Fig. 7 for comparison). A similar property is also observed when the length of n-complete test suites is compared with the upper bound; see, e.g. [9, 19]. The average length of n-complete test suites for the FSMs used in our experiment exceeds 3000, whereas the average length of p-complete test suites, for $p = n - 1$, is less than 700, i.e. they are at least four times shorter than n-complete test suites. In the next experiment, we investigate the fault coverage of p-complete test suites.

By definition, n-complete test suites provide 100% fault coverage in the fault domain $\Im_n$. As p-complete test suites are shorter, they are expected to provide lower fault coverage. Since the number of FSMs in $\Im_n$ is huge (for the FSMs in our experiment, the fault domain $\Im_n$ has as many as 200^{200} FSMs), we estimate the fault coverage using a mutation approach [21]. Given a specification FSM M, a mutant is generated by changing the end states of randomly selected transitions of M. Outputs of transitions are not mutated, since output faults are rather easy to catch, thus not very interesting for fault coverage analysis. Note that the higher the number of mutated transitions, the bigger the difference between the mutant and the specification and, consequently, the higher the probability for the mutant to be killed, i.e. to be T-distinguishable from M. Thus, various percentages of mutated transitions are considered. Mutants that are equivalent to M are discarded since they are not relevant to the estimation of the fault coverage. For each FSM and each p-complete test suite, we generated 10 000 FSMs by mutating k% transitions of M, for $k = 0.5, 2, 5, 10$. We then determined

[2] Each p-complete test suite was generated in less than 0.01 s on an Intel 2.4 GHz computer running Gentoo linux.

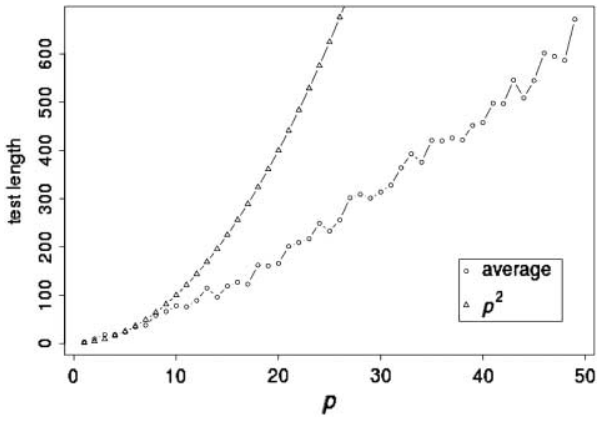


FIGURE 7. Average length of p-complete test suites.

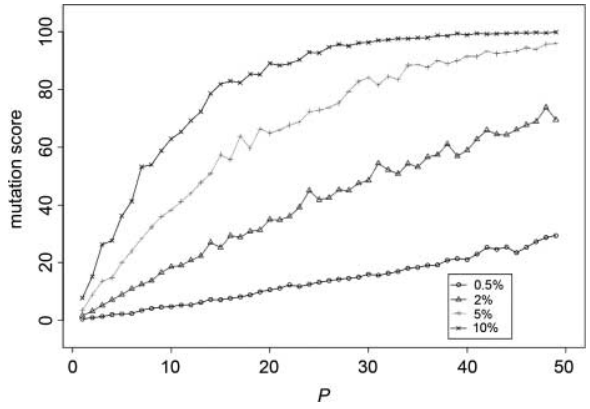


FIGURE 8. Fault coverage of p-complete test suites.

the mutation score, i.e. the percentage of mutants killed by each p-complete test suite. The variation of the score mutation with respect to p is shown in Fig. 8. Note that for mutants with 0.5% of mutated transitions, the fault coverage of p-complete test suites grows linearly; however, it does not exceed 30%. As expected, the coverage improves with the number of mutated transitions. For instance, for mutants with 10% of mutated transitions, a high fault coverage of 95% is obtained with an $(n/2)$-complete test suite.

The experimental results indicate that the proposed approach for incremental test generation of tests parameterized with p reaching the number of states n in the specification FSM can be used for relatively complex specifications. Its effectiveness in fault detection lies in the possibility of using shorter than n-complete tests, which nevertheless provide good fault coverage for 'buggy' implementations, increasing the value of p to enlarge the tests until an implementation bug is discovered, and providing at the end of the testing process well-defined guaranteed fault coverage in terms of the number of states in implementations.

8. CONTRIBUTIONS AND RELATED WORK

In this section, we summarize the contributions of this paper, comparing them with the related work in four research directions.

First, test generation for a fault domain containing only implementation FSMs with fewer states than the specification FSM is investigated, addressing the concern of the scalability of complete tests for sizeable specifications. Note that all the existing methods for complete test suite generation provide guaranteed fault coverage only for fault domains that necessarily include FSMs with at least as many states as in a specification FSM. As a result, they offer no means to avoid a test explosion, while the proposed approach allows the test designer to find a compromise between the guaranteed fault coverage and the size of a test suite.

Second, the proposed approach allows incremental test generation; it may start not with a trivial test suite as all the existing methods, but with some tests already conceived by the test designer. The problem of test extension has in fact been considered in previous work, namely, [22, 23]. However, these methods assume that an existing test suite is n-complete for a given specification FSM M that is modified into another FSM. Thus, tests have to be added to the test suite until a test suite complete for the modified machine is obtained. The method of [22] assumes further that the parts of the implementation that correspond to the unmodified parts of the specification have not been changed. The approach of [23] relies on the knowledge of not only a method that produced the initial test suite, but also the state identification sequences used in it. In the setting assumed in this paper, no such assumptions are needed.

Third, the proposed test generation method improves the existing methods that start with a trivial test suite and terminate with an n-complete test suite (aka checking experiments). These methods rely on 'centralized' state identification, in the sense that all sequences that distinguish a state in question from all the other states are applied after a transfer sequence chosen to reach the state (the reader is referred to several surveys available, e.g. [5, 7]). This is achieved without using the reset input, when there exists a preset distinguishing sequence, as in [24], or an adaptive one, as in [5, 25]. However, when preset or adaptive distinguishing sequences cannot be found, characterization sets, i.e. state identifiers containing several sequences, and the reset input are usually used to ensure that all of them extend the same transfer sequence from a chosen state cover. Different from these methods, the proposed method allows state identification in a 'distributed' way, meaning that sequences in a state identifier, distinguishing a given state from all the other states, can in fact extend not necessarily the same but various convergent transfer sequences for this state. As a result, not only state identifiers, as in [12], but also transfer sequences can be chosen *on-the-fly*, while an n-complete test suite is constructed. Thus the method exploits new possibilities for overlapping subsequences in a complete test suite and reducing

its length. Moreover, since the reset input is not necessarily used each time a given state is to be reached, the number of tests in a test suite, i.e. the number of reset inputs, can thus become a subject for optimization. Examples in Section 6 illustrate a potential saving which the proposed method can achieve.

Fourth, the proposed algorithm improves the state-of-the-art in fault coverage analysis. The sufficient conditions for the p-completeness proposed in this paper generalize the existing ones, such as [7, 8, 12, 21], by allowing $p < n$ and further relax them for the case $p = n$. In our recent work [18], we elaborated sufficient conditions for the case of $p = n$ and showed that they are weaker than the sufficient conditions in [12] for checking experiments and those in [24] for checking sequences. Besides being applicable when $p < n$, the conditions proposed in this paper require the existence of an initialized convergence-preserving transition cover, while in [18], not only the convergence, but also divergence is considered for the tests in the initialized transition cover. Moreover, the conditions rely on new possibilities for determining divergence and convergence of tests, which are not used in the previous work. Thus, the formulated sufficient conditions are weaker than the existing ones.

9. CONCLUSION

In this paper, we considered a problem of incrementally generating tests until the desired level of fault coverage is reached. Solving this problem, we presented sufficient conditions for test suite completeness that are weaker than the ones known in the literature. Based on these conditions, we proposed an algorithm that generates a test suite with complete fault coverage starting with a given set of initial tests, if it is available. The algorithm determines whether the initial test suite already satisfies the sufficient conditions and, thus, can also be used for test suite analysis. The possibility of augmenting the fault coverage of test suites also demonstrates the fact that the algorithm allows one to generate tests using specification coverage as well as fault coverage criteria. Note that these two criteria are often considered as alternatives, where specification coverage criteria are presumed to be more practical. Finally, we experimentally compared both the length and fault coverage of p-complete test suites, for $p < n$, with those of n-complete ones; the results suggest that a trade-off between the test length and the fault coverage can be obtained by selecting a proper value of p.

As a forthcoming step in this work, it is interesting to investigate how the results in this paper can be extended to other fault domains, e.g. to deal with cases when the implementations may have more states than the specification.

FUNDING

The authors acknowledge financial supports of the Natural Sciences and Engineering Research Council of Canada (Grant OGP0194381) and the Brazilian Funding Agency Conselho Nacional de Desenvolvimento Científico e Tecnológico (Grant 200032/2008-9).

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their useful comments.

REFERENCES

[1] Hennie, F.C. (1965) Fault-Detecting Experiments for Sequential Circuits. *Proc. Fifth Annual Symp. Circuit Theory and Logical Design*, Princeton, USA, November 11–13, pp. 95–110.

[2] Vasilevskii, M.P. (1973) Failure diagnosis of automata. *Cybernetics*, **4**, 653–665.

[3] Chow, T.S. (1978) Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, **4**, 178–187.

[4] Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M. and Ghedamsi, A. (1991) Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, **17**, 591–603.

[5] Lee, D. and Yannakakis, M. (1996) Principles and methods of testing finite state machines—a survey. *Proc. IEEE*, **84**, 1090–1123.

[6] Petrenko, A. and Yevtushenko, N. (2005) Testing from partial deterministic FSM specifications. *IEEE Trans. Comput.*, **54**, 1154–1165.

[7] Petrenko, A., Bochmann, G.v. and Yao, M. (1996) On fault coverage of tests for finite state specifications. *Comput. Netw. ISDN Syst.*, **29**, 81–106.

[8] Binder, R. (2000) *Testing Object-Oriented Systems*. Addison-Wesley, Reading, MA.

[9] Simão, A. and Petrenko, A. (2009) Comparing FSM test coverage criteria. *IET Softw.*, **3**, 91–105.

[10] Nebut, C., Fleurey, F., Traon, Y. and Jezequel, J. (2006) Automatic test generation: a use case driven approach. *IEEE Trans. Softw. Eng.*, **32**, 140–155.

[11] Fraser, G., Weiglhofer, M. and Wotawa, F. (2008) Coverage Based Testing with Test Purposes. *Proc. Eighth Int. Conf. Quality Software*, Oxford, UK, August 12–13, pp. 199–208. IEEE Computer Society.

[12] Dorofeeva, R., El-Fakih, K. and Yevtushenko, N. (2005) An Improved Conformance Testing Method. *Formal Techniques for Networked and Distributed Systems*, Taipei, Taiwan, October 2–5, pp. 204–218. *Lecture Notes in Computer Science* 3731. Springer, Berlin.

[13] Karp, R.M. (1976) The Probabilistic Analysis of Some Combinatorial Search Algorithms. In Traub J.F. (ed.), *Algorithms and Complexity: New Directions and Recent Results.* Academic Press, New York. pp. 1–19.

[14] Griggs, J.R. (1983) Lower bounds on the independence number in terms of the degrees. *J. Comb. Theory B*, **34**, 22–39.

[15] Jagota, A. and Sanchis, L.A. (2001) Adaptive, restart, randomized greedy heuristics for maximum clique. *J. Heuristics*, **7**, 565–585.

[16] Wang, R.L., Tang Z. and Cao, Q.P. (2003) An efficient approximation algorithm for finding a maximum clique using Hopfield network learning. *Neural Comput.*, **15**, 1605–1619.

[17] Galil, Z. and Italiano, G.F. (1991) Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, **23**, 319–344.

[18] Simão, A. and Petrenko, A. (2007) Checking FSM Test Completeness Based on Sufficient Conditions. CRIM-07/10-20, Montreal, QC, Canada.

[19] Dorofeeva, R., Yevtushenko, N., El-Fakih, K. and Cavalli, A. (2005) Experimental Evaluation of FSM-Based Testing Methods. *Third IEEE Int. Conf. Software Engineering and Formal Methods*, Koblenz, Germany, September 7–9, pp. 23–32. IEEE Computer Society.

[20] Sidhu, D.P. and Leung, T. (1989) Formal methods for protocol testing: a detailed study. *IEEE Trans. Softw. Eng.*, **15**, 413–426.

[21] DeMillo, R.A., Lipton, R.J. and Sayward, F.G. (1978) Hints on test data selection: help for the practicing programmer. *Computer*, **11**, 34–41.

[22] El-Fakih, K., Yevtushenko, N. and Bochmann, G.v. (2004) FSM-based incremental conformance testing methods. *IEEE Trans. Comput.*, **30**, 425–436.

[23] Pap, Z., Subramaniam, M., Kovacs, G. and Nemeth, G.A. (2007) A Bounded Incremental Test Generation Algorithm for Finite State Machines. *TestCom/FATES 2007*, Tallinn, Estonia, June 26–29, pp. 244–259. *Lecture Notes in Computer Science* 4581. Springer, Berlin.

[24] Ural, H., Wu, X. and Zhang, F. (1997) On minimizing the lengths of checking sequences. *IEEE Trans. Comput.*, **46**, 93–99.

[25] Kohavi, Z., Rivierre, J.A. and Kohavi I. (1973) Machine distinguishing experiments. *Comput. J.*, **16**, 141–147.

Apêndice E

A. S. Simão, A. Petrenko. and Maldonado, J. C.
Comparing Finite State Machine Test Coverage
Criteria. IET Software, v. 3, p. 91-105, 2009

Published in IET Software
Received on 27th February 2008
Revised on 19th November 2008
doi: 10.1049/iet-sen.2008.0018

In Special Issue on Selected papers from SBES '07

ISSN 1751-8806

Comparing finite state machine test coverage criteria

A. Simão[1] *A. Petrenko*[2] *J.C. Maldonado*[1]

[1]*Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, São Paulo, Brazil*
[2]*Centre de Recherche Informatique de Montreal (CRIM), Montreal, Quebec, Canada*
E-mail: adenilso@icmc.usp.br

Abstract: To plan testing activities, testers face the challenge of determining a strategy, including a test coverage criterion that offers an acceptable compromise between the available resources and test goals. Known theoretical properties of coverage criteria do not always help and, thus, empirical data are needed. The results of an experimental evaluation of several coverage criteria for finite state machines (FSMs) are presented, namely, state and transition coverage; initialisation fault and transition fault coverage. The first two criteria focus on FSM structure, whereas the other two on potential faults in FSM implementations. The authors elaborate a comparison approach that includes random generation of FSM, construction of an adequate test suite and test minimisation for each criterion to ensure that tests are obtained in a uniform way. The last step uses an improved greedy algorithm.

1 Introduction

Model-based testing refers to the derivation of test suites from a model representing software behaviour. Such models can be constructed early in the development cycle, allowing testing activities to start before the coding phase, as tests can be based on what the software should do, and not on what the software does. Finite state machines (FSMs) are state-based models, which have been widely used in many areas, such as hardware design, language recognition, conformance testing of protocols and object-oriented software testing; for example, [1, 2]. The existence of several methods for test generation from state-based models provides flexibility for testers to devise effective testing strategies.

Test generation methods are based on coverage criteria. A coverage criterion defines a set of testing requirements that must be covered by an adequate test suite. It is usually derived from elements of the model that the tester considers important to be tested. For instance, a coverage criterion can require that all transitions of an FSM must be traversed. There exist several coverage criteria that can be used to guide test generation, as well as to assess the quality of a given test suite. Usually, the cost of a coverage criterion can be estimated by the length of a test suite that is required to satisfy it. When one has to choose among several coverage criteria, it is desirable to use the most effective applicable criterion, that is, the criterion that has the highest probability to reveal the faults in the implementation under test with a minimum cost. A high fault detection capability usually comes with a price: the tests may simply explode and then a weaker criterion might be used instead. Budget and schedule constraints must also be taken into account. For instance, if the tests are manually executed, their total length should be much shorter than those executed automatically. Therefore it is important to be able to estimate the length of tests adequate for various testing criteria.

The comparison of test coverage criteria can be based on their theoretical properties, for example, upper bounds for test lengths and subsumption relations [3]. As an example, Binder [2] discusses the trade-offs of various state-based test strategies, highlighting the importance of comparing the expected length of test suites generated by different approaches when a test strategy must be chosen. The discussion is based on the worst-case minimum and maximum lengths. However, the maximum lengths are reached for FSMs with a special structure, for example,

Moore lock FSMs that require the longest sequence to reach and identify a certain state [4]. Thus, the usage of upper bounds for various coverage criteria can be misleading. It is important to have at least some indications on the average lengths of adequate test suites. Based on these indications, a test engineer can plan a testing strategy that better fits the constraints of a testing project. Concerning the subsumption relation [3], which indicates when a test suite adequate to one criterion is also adequate to another, it can be established for some criteria; however, not all of them are comparable with respect to this relation. Then, it is important to have other means of comparing such criteria. In this context, experimental data are useful for choosing coverage criteria and defining effective testing strategies. Experimental data characterising the average lengths of test suites adequate for various criteria help in assessing the applicability of a particular criterion. Furthermore, assuming the tester has chosen a given criterion, an important question is how the test suites adequate for this criterion relate to others in order to know how the cost would change if the tester decides to generate a test suite that is adequate according to another stronger criterion.

Despite the importance of experimental data, there is a lack of work in the literature that provides those concerning FSM tests. The monograph [2] refers to just the worst-case test lengths. We are aware of only the work of Dorofeeva *et al.* [5], which reports the results of an experiment comparing various test generation methods. However, no experimental comparison among coverage criteria for FSMs is available. In this paper, we address the experimental comparison of test coverage criteria for FSMs. The contributions of this paper are 2-fold. First, we consider four criteria, namely, state coverage (SC), transition coverage (TC), initialisation fault (IF) coverage and transition fault (TF) coverage criteria, and provide experimental data on the length of tests generated from an FSM specification to satisfy these coverage criteria. We investigate the impacts of FSM parameters on the cost associated with the usage of those criteria. Although the cost of test suites adequate for various criteria can be estimated in various ways, we use the length of test suites as a measure of the cost since it is an objective measure which can easily be obtained for a large number of FSMs, as required in our experiments, and provides a good approximation of the real cost: all things being equal, longer test suites are likely more expensive. Thus, we are interested in comparing the average length of the test suites for those criteria, both to each other and to the theoretical upper limits. We also investigate how the test suites adequate for these criteria are related to the notion of n-completeness [6], which plays an important role in the comparison of test generation methods. The experiments involve random generation of FSM specifications and tests in order to provide experimental characterisation of how the test length depends on FSM parameters and coverage criteria.

Secondly, we elaborate the approach for comparing criteria, which ensures that tests are generated in a uniform way. This is achieved by first constructing a test suite adequate for all the criteria and minimising it for each criterion with a generalised greedy algorithm. We propose a heuristics that decreases the execution time of the algorithm, without compromising much its effectiveness.

The paper is organised as follows. Section 2 contains basic definitions related to FSMs and test suites. In Section 3, we present the main concepts related to test coverage criteria and define the criteria that we investigate in this paper. The discussion on how to compare the cost of different criteria based on the length of the adequate test suites is presented in Section 4. Section 5 details the comparison approach which includes random generation of FSM, construction of adequate test suite and test minimisation. The results of the experiments and their analyses are presented in Section 6. In Section 7, we discuss the threats to the validity of the results. Finally, in Section 8, we draw concluding remarks and point to future work.

2 FSM and tests

An FSM is a deterministic Mealy machine, which can be defined as follows.

Definition 1: An FSM M is a 7-tuple $(S, s_0, I, O, D, \delta, \lambda)$, where

- S is a finite set of states with the initial state s_0,
- I is a finite set of inputs,
- O is a finite set of outputs,
- $D \subseteq S \times I$ is a specification domain,
- δ: $D \rightarrow S$ is a transition function and
- λ: $D \rightarrow O$ is an output function.

An FSM M is said to be completely specified (a complete FSM, CFSM), if $D = S \times I$. Otherwise, M is called a partially specified machine (a partial FSM, PFSM). A tuple $(s, x) \in D$ is a transition of M. Fig. 1*a* presents an example of a partial FSM. The initial state is highlighted in bold. The input symbols are a and b, and the output symbols are 0 and 1. The label 'x/y' of an edge (transition) from state s to state s' indicates that $\delta(s, x) = s'$ and $\lambda(s, x) = y$, that is, when the machine is in state s, it responds to input x by producing output y and moving to state s'. State s' is the tail state of this transition.

A string $x_1, \ldots, x_k \in I^*$ is said to be a defined input sequence at state $s \in S$ if there exist $s_1, \ldots, s_k, s_{k+1}$, where $s_1 = s$, such that $(s_i, x_i) \in D$ and $\delta(s_i, x_i) = s_{i+1}$ for all $i = 1, \ldots, k$. We use $\Omega_M(s)$ to denote the set of all defined input sequences for state s and Ω_M as a shorthand for $\Omega_M(s_0)$, that is, for the input sequences defined for the initial state of M and, hence, for M itself. Given sequences $\alpha, \beta \in I^*$, we write

doi: 10.1049/iet-sen.2008.0018

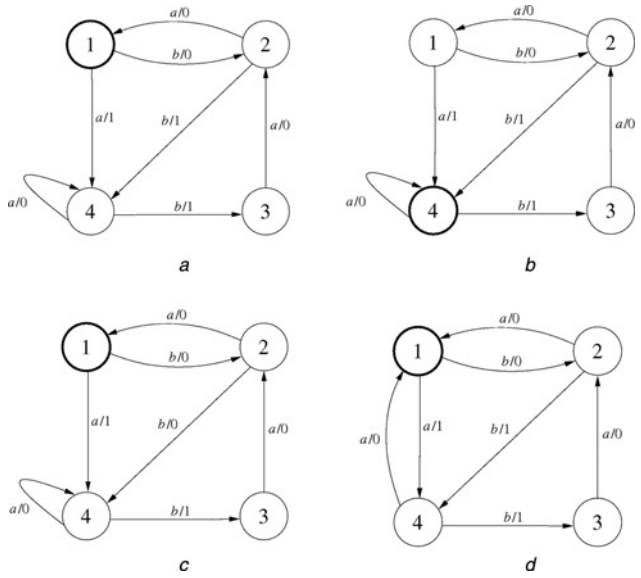


Figure 1 Partial FSM and some of its mutants

a Partial FSM
b Initialisation fault mutant
c Output fault mutant
d Transfer fault mutant

$\alpha \leq \beta$, if α is a prefix of β. For a sequence $\beta \in I^*$, pref(β) is the set of prefixes of β, that is, pref(β) = $\{\alpha \mid \alpha \leq \beta\}$. For a set of sequences $T \subseteq I^*$, pref(T) is the union of pref(β), for all $\beta \in T$; T is prefix-closed if T = pref(T).

We extend, as usual, the transition and output functions from input symbols to defined input sequences. For the empty sequence ε, we have that $\delta(s, \varepsilon) = s$ and $\lambda(s, \varepsilon) = \varepsilon$ for any $s \in S$. For an input sequence α defined at a state $s \in S$ and an input x, we have that $\delta(s, \alpha x) = \delta(\delta(s, \alpha), x)$ and $\lambda(s, \alpha x) = \lambda(s, \alpha)\lambda(\delta(s, \alpha), x)$. A sequence $\alpha \in \Omega_M$ is a transfer sequence to a state s, if $\delta(s_0, \alpha) = s$. An FSM M is said to be initial connected if for each state $s \in S$ there exists a transfer sequence to s. In this paper, we assume that FSMs for which tests are generated are initially connected, since any state that is not reachable from the initial state can be removed without changing the behaviour of the machine. A natural r is called an accessibility degree of the FSM M if for each state there exists a transfer sequence to the state with at most r input symbols.

Given an FSM $M = (S, s_0, I, O, D, \delta, \lambda)$, states s and t are distinguishable, denoted by $s \not\sim t$, if there exists an input sequence $\gamma \in \Omega_M(s) \cap \Omega_M(t)$, such that $\lambda(s, \gamma) \neq \lambda(t, \gamma)$; γ is called a separating sequence for s and t. A natural d is called a distinguishability degree of the FSM M if for any two distinguishable states there exists a separating sequence with at most d input symbols. An FSM is reduced, if all state pairs are distinguishable.

Definition 2: A defined input sequence of FSM M is called a test case (or simply a test) of M. A test suite of M is a finite set of tests of M, such that no test is a proper prefix of another test.

To model implementation faults, we use the notion of a mutant of a given specification FSM.

Definition 3: Given a specification FSM $M = (S, s_0, I, O, D, \delta, \lambda)$, a mutant of M is any FSM over the state set S and input set I.

A mutant $N = (S, s_0', I, O, D_N, \Delta, \Lambda)$ is distinguishable from M, denoted $N \not\sim M$, if there exists $\gamma \in \Omega_M \cap \Omega_N$ such that $\lambda(s_0, \gamma) \neq \Lambda(s_0', \gamma)$. We say that γ kills N. N has a transfer fault in the transition $(s, x) \in D$ with respect to M, if $\delta(s, x) \neq \Delta(s, x)$. N has an output fault in the transition $(s, x) \in D$ with respect to M, if $\lambda(s, x) \neq \Lambda(s, x)$. N has an IF with respect to M, if $s_0 \neq s_0'$. N has a TF in $(s, x) \in D$ with respect to M, if it has an output or transfer fault or both. Fig. 1 shows examples of mutants with each of these faults. The mutant in N has an IF, since the initial state is changed to state 4. The mutant in Fig. 1*c* has an output fault in the transition (2, *b*), since the output was changed from 1 to 0. The mutant in Fig. 1*d* has a transfer fault in the transition (4, *a*), since the tail state of the transition was changed from state 4 to state 1.

Many methods generate test suites that are guaranteed to reveal any possible fault in the implementation under test (under some assumptions). The key property of these test suites is established in the following definition. Recall that, as a mutant is an FSM, a complete mutant is a completely specified FSM.

Definition 4: Let T be a test suite of a reduced FSM M with n states. T is n-complete, if each complete mutant of M with at most n states, which is distinguishable from M, is killed by some test case in T.

Note that the definition refers only to complete mutants. The rationale is that the implementation of an FSM cannot 'refuse' inputs and, thus, they are modelled by completely specified machines.

3 Test coverage criteria

A test coverage criterion can be thought of as a systematic way of defining testing requirements, which an adequate test suite must fulfil. Therefore we can compare two test suites with respect to a given criterion by analysing the set of testing requirements they satisfy. Let K be a test coverage criterion. We use $TR_K(M)$ to denote the set of testing requirements that the criterion K defines for a given FSM M. Let T be a test suite. We define $TS_K(M, T) \subseteq TR_K(M)$ as the set of testing requirements that are satisfied by T. The test coverage of T, denoted by $C_K(M, T)$, is the ratio between the number of testing requirements it fulfils and the total number of testing requirements, that is, $C_K(M, T) = |TS_K(M, T)|/|TR_K(M)|$.

If $TS_K(M, T) = TR_K(M)$, it is said that T is K-adequate for M. A criterion K subsumes another criterion K', if any K-adequate test suite is also K'-adequate.

Test coverage criteria are usually defined with specification or fault coverage in mind. When an FSM is the specification for testing, tests covering an FSM specification target one or several elements such as inputs, outputs, states and fragments of its transition graph. Covering inputs and outputs is usually considered as extremely weak requirement for FSM testing and hence, we will not consider them in this paper. Paths are typical fragments of the transition graph considered for coverage. However, path coverage has to be selective, as the number of paths is infinite in the presence of cycles. One of the most cited criteria is the TC, which we consider in this paper. It is a special case of an 'x-switch' coverage criterion, proposed in [7], which defines a testing requirement as a tuple of transitions to cover by a test; for simplicity, we concentrate only on the traditional TC criterion defined below.

Testing with fault coverage in mind relies on fault models. Fault models represent the kind of faults the tester is interested in at a particular moment. They are important to make the test activity more manageable, aiding with focusing the testing efforts in the particular kind of faults they embody. Among simple FSM fault models, we should mention the IFs and TFs considered in this paper. The former states that the only possible faults in FSM implementations are related to a wrong initial state of a specification FSM, whereas the latter assumes that implementation faults occur in transitions.

Thus, we choose the following four FSM test coverage criteria: (i) SC, (ii) TC, (iii) IF coverage and (iv) TF coverage. These criteria are defined in the next sections.

3.1 State coverage (SC) criterion

For the SC criterion, we assume that reaching a state of the FSM M by some test is a testing requirement. To simplify the presentation, we define $TR_{SC}(M) = S$, where $TR_{SC}(M)$ is a set of states that are required to be covered, whereas S denotes the set of states. A more general way of defining it would be to use a subset of states (to reach and, thus, to cover by tests) instead of the whole set S. $TS_{SC}(M, T)$ is the set of states that are covered by T, and thus, $C_{SC}(M, T) = |TS_{SC}(M, T)|/|S|$. As an example, for the FSM in Fig. 1*a* the test suite {*ab*, *b*} is SC-adequate; note that the initial state is reachable with the empty transfer sequence, while the prefix *a* of the test *ab* is a transfer sequence to state 4.

3.2 Transition coverage (TC) criterion

For the TC criterion, we assume that covering a transition of the FSM M is a testing requirement. Again, for simplicity, we define $TR_{TC}(M) = D$. $TS_{TC}(M, T)$ is the set of transitions covered by tests in T, that is, $TS_{TC}(M, T) = \{(s, x) \in D \mid \exists\ \pi \in T,\ \alpha x \leq \pi,\ \delta(s_0, \alpha) = s\}$. Note that, since only initially connected FSMs are considered, each state can be reached and therefore each transition can be covered. Thus, $C_{TC}(M, T) = |TS_{TC}(M, T)|/|D|$. If T is TC-adequate, then it is easy to verify that T is also SC-adequate. Therefore the TC criterion subsumes the SC criterion. The usefulness of this criterion is that a TC-adequate test suite detects all output faults in implementations, provided that there are no transfer faults. For our example FSM in Fig. 1*a*, the test suite {*aa*, *aba*, *ba*, *bb*} is TC-adequate.

3.3 Initialisation fault (IF) coverage criterion

For the IF coverage criterion, we define coverage with respect to IFs, that is, the testing requirements address the states that could wrongly be used as the initial state of an FSM implementation. To satisfy such a requirement, a test suite should include a sequence which is applied to the suspected initial state and seperates it from the actual initial state. Then, we define $TR_{IF}(M) = \{s \in S \mid s \not\sim s_0\}$. Note that $TR_{IF}(M)$ ranges from the empty set for M with no distinguishable states to $S\backslash\{s_0\}$ for a reduced M. The criterion is, thus, applicable to an FSM with at least one state distinguishable from the initial state. $TS_{IF}(M, T)$ is defined as follows

$TS_{IF}(M, T) = \{s \in S \mid s \not\sim s_0,\ \exists\ \pi, \chi \in T,\ \gamma \leq \pi, \beta\gamma \leq \chi,\ \delta(s_0, \beta) = s,\ \lambda(s_0, \gamma) \neq \lambda(s,\gamma)\}$, and thus, $C_{IF}(M, T) = |TS_{IF}(M, T)| / |TR_{IF}(M)|$.

In this formula, γ is a sequence that distinguishes s_0 from a state s and, hence, the test suite T should contain a test, that starts with γ, as well as a test, that takes the FSM M into the state s and then continues with γ. An IF-adequate test suite should have such tests for each state distinguishable from the initial state. Thus, for reduced FSMs, a test suite that is IF-adequate is also SC-adequate, that is, the criterion IF subsumes the criterion SC. For the FSM in Fig. 1*a*, the test suite {*aa*, *aba*, *bb*} is IF-adequate. Indeed, we have that the input sequence *b* is a transfer sequence to state 2 and is followed by *b*, which distinguishes the initial state and state 2. Similarly, the tests *aba* and *aa* satisfy the requirements related to states 3 and 4, respectively.

3.4 Transition fault (TF) coverage criterion

For a pair $(s, x) \in D$, we define a coverage with respect to TFs, by considering that the transition from state s under input x in some mutant may have an unexpected output or/and wrongly end in another state distinguishable from $\delta(s, x)$. Thus, the set of testing requirements is defined as $TR_{TF}(M) = \{(s, x, s') \in D \times S \mid \delta(s, x) \not\sim s'\}$. Since $TR_{TF}(M)$ is empty for M with no distinguishable states, the criterion is applicable for an FSM with at least one pair of distinguishable states. Thus, a testing requirement is a

doi: 10.1049/iet-sen.2008.0018

pair of a transition, represented by the pair (s, x), and a state from which the tail state of the transition should be distinguished. To satisfy such a requirement, a test suite should not only cover a transition as in the case of the TC criterion, but also have corresponding separating sequences applied in both concerned states. $TS_{TF}(M, T)$ is defined by the requirements that are satisfied

$$TS_{TF}(M, T) = \{(s, x, s') \in D \times S \mid \delta(s, x) \neq s', \exists \pi, \chi \in T, \alpha x \gamma \leq \pi, \beta\gamma \leq \chi, \delta(s_0, \alpha) = s, \delta(s_0, \beta) = s', \lambda(\delta(s_0, \alpha x), \gamma) \neq \lambda(s', \gamma)\}.$$

Thus, $C_{TF}(M, T) = |TS_{TF}(M, T)|/|TR_{TF}(M)|$. For the example FSM (Fig. 1*a*), the test suite {*aaaaa*, *abaaa*, *baa*, *bbaaa*} is TF-adequate. Consider, for instance, the transition (2, *b*), whose tail state is 4. Test *bb* covers this transition. States 4 and 1 are distinguished by *a*, which follows *bb* and the empty sequence; states 4 and 2 are distinguished by *aa*, which follows *bb* and *b*; and states 4 and 3 are distinguished by *aaa*, which follows *bb* and *ab*. Thus, all requirements related to transition (2, *b*) are satisfied. One can check that the other requirements are also satisfied.

We note that the idea of this criterion is similar to the one proposed in [6], where the fault coverage of a given test suite is defined as the percentage of states that are distinguished from the tail state of each transition by the test suite.

For reduced FSMs, if a test suite is TF-adequate, then it is also TC-adequate, since the test suite must cover each transition in order to reveal each transfer fault. Therefore, the criterion TF subsumes the criterion TC, and consequently, SC, for reduced FSMs, as shown in Fig. 2. The criterion TF also subsumes the criterion IF, once the former is augmented with the requirement that all the sequences separating the initial state from a tail state of each transition are appended to the empty sequence.

4 Comparing adequate tests

The definition of testing strategies requires a careful analysis of the cost and benefits of all applicable coverage criteria. This analysis can be based on the known theoretical properties of the criteria. For instance, one may prefer to choose a criterion most powerful in revealing faults. However, if the chosen criterion requires an adequate test suite that is impractical (because of test explosion) or too costly to execute, it will hardly be chosen. Thus, in many practical situations, the cost of applying a criterion becomes

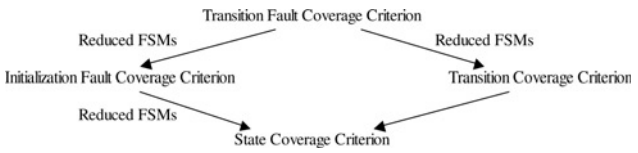


Figure 2 *Subsumption relation of FSM coverage criteria*

a major factor in choosing a proper test coverage criterion. For simplicity, we assume here that the total length of an adequate test suite with respect to a given criterion is the cost of the test suite and, thus, the cost of applying the criterion for a given specification FSM. Although this measure neglects important practical issues about the execution of a test suite, such as a varying cost of executing different inputs, it provides a fair basis for comparing different criteria. For instance, distinct costs could be represented by weighted inputs, but the impact of these weights should be uniform among the criteria. Thus, we assume that all inputs have equal cost.

The upper bounds of the test length for the criteria that we are considering grow rapidly with the FSM parameters (see discussions below); these bounds characterise the so-called test explosion effect. Although at least some of these bounds are shown to be tight, we want to know if the notorious test explosion may occur for a given FSM and for each coverage criterion and, if it does, how big it might be on an average compared with what the formulae indicate. Ideally, if an FSM specification is available in a machine-processable form and an appropriate FSM test generation tool is easily accessible, one would just generate a test suite for each of the candidate coverage criterion and choose the one that corresponds to a desired compromise between test effectiveness and cost. In reality, however, a number of factors can prevent testers from following this simple-minded method. For example, a test strategy may have to be chosen even before a detailed specification is obtained or tools might not always be readily available. Last but not the least, one may not need to generate an adequate test suite for a given criterion; he may well restrict himself to, for example, '90%' of coverage for a certain criterion. In such situations, experimental data, if available, may provide indications on the expected length of test suites 90%-adequate for the chosen coverage criterion.

The upper bounds of the length of tests adequate for various coverage criteria can be derived by considering an FSM with the 'worst' values of parameters for a given criterion. For the SC criterion, such a parameter is the accessibility degree r, which is the maximum length of a minimal transfer sequence to a given state; clearly, $0 \leq r \leq n - 1$. Henceforth, n denotes the number of states, k the number of inputs and l the number of outputs. The length of SC-adequate test suites does not exceed rn, thus $n(n - 1)$ (note that the formula can further be refined by excluding prefixes of transfer sequences). Similarly, the length of TC-adequate test suite is bounded by $kn(r + 1) = kn^2$. For the initialisation and TF coverage criteria, the distinguishability degree d has also to be taken into account. It may reach the value of $n - 1$ for complete FSMs and $n(n - 1)/2$ for partial FSMs. An IF-adequate test suite may contain $n - 1$ separating sequences applied in the initial state as well as $n - 1$ transfer sequences each of which is followed by a separating sequence. The total length does not exceed $d(n - 1) + (n - 1)(r + d) =$

$(n-1)(n-1+2d)$. Then, complete FSMs may require up to $(n-1)(n-1+2n-2)=3(n-1)^2$, whereas partial FSMs $(n-1)(n-1+2n(n-1)/2)=(n+1)(n-1)^2$. For the TF coverage, a single transition may require at most two tests, each of which does not exceed the value $r+1+d$; hence, kn transitions need $2kn(r+1+d)=2kn(n+d)$ inputs. Thus, the length of the TF-adequate test suite does not exceed $2kn(2n-1)$ for complete FSMs and $kn^2(n+1)$ for partial ones.

In addition to the above characterisation of worst cases, one may also consider asymptotic characterisation of FSM parameters for 'almost all FSMs'. Indeed, the monograph [8] indicates that the accessibility degree r is asymptotically equal to $\log_k n$ and the distinguishability degree d is asymptotically equal to $\log_k \log_l n$ for a complete FSM with n states, k inputs and l outputs. These formulae give the values expected to be valid for almost all FSMs. We use them to derive the expected length of the test suites for the four criteria. For the SC-criterion, the expected length is $rn=n \log_k n$. For the TC-criterion, the length is $kn(r+1)=kn(\log_k n+1)$. The IF-criterion yields (for complete FSMs) $d(n-1)+(n-1)(r+d)=(n-1)(\log_k n+2 \log_k \log_l n)$. Finally, for the TF-criterion, the expected length is $2kn(r+1+d)=2kn(1+\log_k n+\log_k \log_l n)$.

At the same time, given a specification FSM and a coverage criterion, it is not clear how close to these bounds the test length might be. Since currently it does not seem plausible to gather sufficient data about actual specifications and tests adequate for various criteria, experiments involving random generation of specifications and tests may provide experimental characterisation of how the test length depends on FSM parameters and coverage criteria. The remaining part of this paper is devoted to the experiments addressing the following questions:

- How does the average length of an adequate test suite compare with the upper bound?
- How do test suites adequate for various criteria relate in terms of the length?
- If a test suite is adequate for one criterion, how adequate would it be for another criterion?
- Which of the FSM parameters contribute more to test explosion and for which of the four criteria?
- How probable is the condition that test suites adequate for various criteria are n-complete?

5 Comparison approach

Experiments for comparison of testing criteria are based on the following main operations on FSMs and tests: (i) FSM generation, (ii) generation of a test suite adequate for the given criteria and (iii) minimisation of a test suite with respect to a given criterion. In the following sections, we explain these operations.

5.1 FSM generation

We implemented a tool to randomly generate initially connected FSMs with given numbers of states, inputs, outputs and transitions. The tool first generates sets of states, inputs and outputs with the required number of elements. The generation proceeds then in two phases. In the first phase, a state is selected as the initial state and marked as 'reached'. Then, for each state s not marked as 'reached', the generator randomly selects a reached state s', an input x and an output y, adds a transition from s' to s with input x and output y, and marks s as 'reached'. When this phase is completed, an initially connected FSM is obtained. In the second phase, the generator adds, if needed, more transitions (by randomly selecting two states, an input and an output) to the machine until the required number of transitions is obtained.

There are at least two alternatives to the random generation approach. First, one may involve human testers in experiments by asking them to generate FSMs using their experience and domain knowledge. This setting would allow considering the human factor in the experiments and hopefully obtaining more 'realistic' FSM specifications. However, manual generation of a sufficient number of FSMs could be excessively expensive. Another alternative would be to use only FSMs found in the literature, forming a benchmark of FSMs. This setting is attractive, but again, not many such FSMs are publicly available.

5.2 Test generation

To compare the length of test suites implied by various test coverage criteria, one first needs to generate these tests in a uniform way, as the test length may significantly vary depending on algorithms used for test generation. As an example, to derive a test suite adequate for the SC criterion, one may use different graph traversal algorithms, obtaining test suites of different lengths. Similarly, there are various algorithms for generating test sequences for the other criteria. One possibility of reducing any impact of using different search algorithms and enforcing the uniformity of test generation with various criteria is to use only one test generation algorithm that yields a test suite adequate for all the test coverage criteria considered. Once such a 'super' test suite is obtained, one may then determine a (minimal) subset of this test suite adequate for a given criterion and to compare the lengths of the resulting adequate test suites. This approach is implemented as a two-step procedure: (1) generate a (quasi-minimal) test suite adequate for all the four criteria and (2) minimise it for each criterion.

IET Softw., 2009, Vol. 3, Iss. 2, pp. 91–105
doi: 10.1049/iet-sen.2008.0018

For a given specification FSM $M = (S, s_0, I, O, D, \delta, \lambda)$, a test suite which is SC-, TC-, IF- and TF-adequate is generated in the following manner. For each pair of states s and s', we determine a shortest distinguishing input sequence, $\gamma_{s,s'}$. Note that, as non-reduced FSMs can also be generated, there may be some state pairs for which no such sequence exists. Then, we determine a minimal transition cover T by building a spanning tree of M and augmenting it with missing transitions. We add the empty sequence to T. The test suite is initialised with T. Finally, for each $\alpha \in T$, $\delta(s_0, \alpha) = s$ and each $s' \in S$, such that $\delta(s_0, \alpha)$ is distinguishable from s', we include $\alpha\gamma_{s,s'}$ in T. The resulting test suite is n-complete for any reduced FSM M, since the adopted test generation algorithm is in fact the HSI-method [9] developed for reduced FSMs, and the test suite is what we need for our experiments: it is SC-TC-IF-TF-adequate.

5.3 Test minimisation

Given a specification FSM M and an SC-TC-IF-TF-adequate test suite T, we need to determine its subsets adequate for state, transition, initialisation and TF coverage criteria, that is, SC-, TC-, IF- and TF-adequate test suites, respectively.

Thus, the problem of test minimisation arises. Given a test suite T and a particular criterion K, we want to find $T' \subseteq T$ such that $\mathrm{TS}_K(M, T') = \mathrm{TS}_K(M, T)$ and the cost function $w(T')$ is minimised. As a special case, if T is K-adequate, T' is also K-adequate. The cost function can be defined to reflect the cost of applying a given test suite. We define the cost $w(\alpha)$ of a sequence $\alpha \in I^*$ as $|\alpha| + 1$, that is, the length of α plus the implicit reset symbol used to bring the FSM back to the initial state before applying α. We define $w(R)$ as the sum of $w(\alpha)$ of all sequences $\alpha \in R$, such that α is not a proper prefix of another sequence in R. Thus, it is assumed that all inputs are of the same cost, although if needed, one can easily diversify the cost of inputs.

For the SC criterion, we need to find a minimal subset $T' \subseteq T$ that reaches every state of a given FSM. This can be posed as a weighted set-cover problem, where the ground set is the set of states and the covering elements are tests (as well as all their prefixes). This problem is known to be NP-complete [10]. A greedy algorithm can be used to find a near optimal covering set. We start with an empty covering set $T' = \emptyset$. At each step, we pick up a sequence $t \in T \backslash T'$ that is the most cost-effective and include it in T'. The cost-effectiveness of a sequence t with respect to T' is defined as the ratio between the cost and coverage increments induced by the inclusion of t in T', that is, $(w(T' \cup \{t\}) - w(T'))/|\mathrm{TS}_{\mathrm{SC}}(M, T' \cup \{t\}) \backslash \mathrm{TS}_{\mathrm{SC}}(M, T')|$. For the TC criterion, a similar approach can be followed by replacing the set states S by the set of defined transitions D.

For the IF and TF coverage criteria, the test minimisation problem cannot directly be cast as a set-cover problem, since to cover some testing requirements two sequences may be needed at the same time. In this case, the test minimisation problem is defined as a set-cover with pairs (SCP). The SCP problem can be viewed as a generalisation of the classical set-cover problem (see [11] for discussion on its complexity). Hassin and Segev [11] propose a generalisation of the greedy algorithm to work with pairs of elements. At each iteration, the cost-effectiveness of single sequences as well as pairs of sequences is evaluated and the most cost-effective one is selected (either a single sequence or a pair of sequences).

We noted that, although the algorithm checks single sequences and pairs of all the given sequences to determine which are the most cost-effective, in almost all iterations, the algorithm ends up selecting a single sequence, if it exists. In these cases, computing the cost-effectiveness of pairs of sequences usually does not significantly contribute to the results, but it is a resource-consuming process. Therefore we propose below a slightly different algorithm implementing the heuristics that at each iteration the cost-effectiveness of sequence pairs is only computed if no single sequence increases the coverage. In Fig. 3, we compare the proposed algorithm with the original algorithm of [11]. We consider

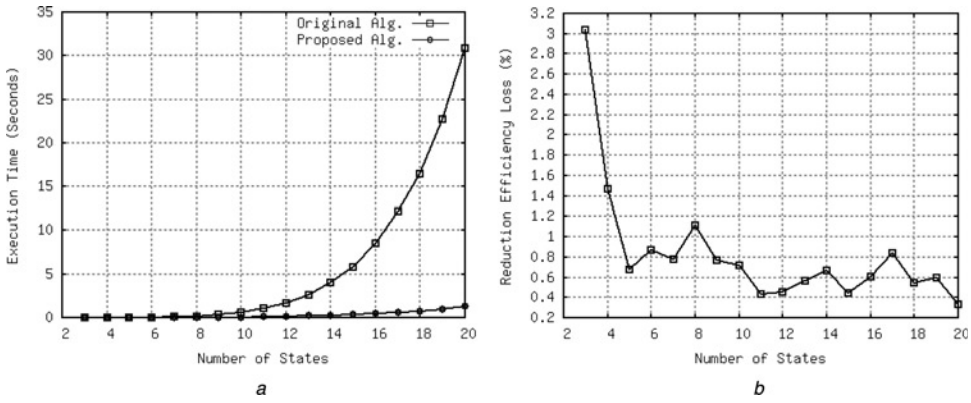


Figure 3 *Proposed algorithm with the original algorithm*

a Average time execution of original and proposed algorithms
b Average loss of reduction efficiency

the reduction ratio and the execution. We randomly generate 100 FSMs with two inputs, two outputs, the number of states ranging from 3 to 20 and the degrees of completeness of 0.4, 0.6, 0.8, 0.9 and 1.0. The degree of completeness is the ratio between the defined transitions and the number of possible transitions in a deterministic FSM, that is, kn. Note that the degree 1.0 corresponds to complete FSMs. Note also that at least $n - 1$ transitions are required for the FSM to be initially connected. Therefore we subtract $n - 1$, both from the number of defined transitions and the number of possible transitions. Thus, given the number of transitions t, the degree of completeness m is calculated as $m = (t - n + 1)/(kn - n + 1) = (t - n + 1)/(k(n - 1) + 1)$. For instance, for an FSM with $k = 4$ and $n = 20$ and with 55 transitions, the degree of completeness is 0.59. To obtain an FSM with a given degree of completeness m, we calculate the number of transitions that is necessary to guarantee that the degree of completeness is at least m. An SC-TC-IF-TF-adequate test suite is obtained for each FSM and then minimised with respect to the transfer fault coverage criterion. Note that state and TC criteria would not be useful for this comparison, since they require only a single sequence to cover a testing requirement. An IF coverage criterion could be used as well. Fig. 3*a* shows the average time required to execute the algorithms. The execution time for FSMs with up to ten states is comparable for both algorithms. However, for larger FSMs, the execution time of the original algorithm increases quicker than that for the proposed algorithm. In Fig. 3*b*, we present the loss of reduction efficiency. Let t_o and t_n be the length of the test suite obtained by the original algorithm and the proposed one, respectively. Then, the reduction efficiency loss is determined as $(t_n - t_o)/t_o$, that is, the percentage of the additional length of the test suites produced by the proposed algorithm with respect to the original algorithm. We can observe that, although for FSMs with three states the reduction loss is about 3%, for FSMs with at least five states, the reduction loss is about 1%. Therefore considering the reduction of the execution time, the decrease in the efficiency in the proposed algorithm is rather low.

We further generalised the algorithm to deal not only with pairs, but also with arbitrary subsets of sequences in order to minimise test suites using more complex criteria. In the algorithm, the variable p indicates a size of the considered subsets. In each iteration, the value of p is initially set to 1 and incremented until there exists a set of p sequences, which increases the coverage of requirements.

Note that it is not necessary to define a maximum value of p in the algorithm. Indeed, for any of the criteria defined in this paper, there exists a maximum value of p which represents the largest number of sequences necessary to cover a single requirement. For state (respectively, transition) coverage, a single sequence is sufficient to cover a state (respectively, a transition). Thus, the maximum value of p is 1. For transition and IF coverage criteria, the maximum value of p is 2, since in some cases, two sequences might be needed to cover a single requirement. Observe that one can formulate complex test coverage criteria, which, for example, concern simultaneously several paths, requiring multiple sequences to cover. Nonetheless, the generalised greedy algorithm could be used to minimise a test suite based on these criteria as well. We observe that, if the value of p is limited to 1, the algorithm is an instantiation of the classical greedy algorithm for the set-cover problem, and for $p = 2$ of the algorithm for SCP in [11]. The difference, as stated earlier, is that, to accelerate computations, the coverage of sequence pairs is computed as a last resort. Moreover, our notion of cost-effectiveness of covering elements differs from that in [11], since we need to take into account the relation 'is a prefix' between tests, so that, for example, including into a cover a longer test after its prefix has already been included is not 'penalised'. At the same time, our test minimisation algorithm (Fig. 4) needs to determine at each iteration the set of covered testing requirements (ground elements), a trivial step in the abstract set-cover problem, which may become involved for a complex test criterion. As an example, consider the complexity of determining $\mathrm{TS}_{\mathrm{TF}}(M, T)$. For each pair of sequences $\alpha, \beta \in T$, and for each requirement $(s, x, s') \in \mathrm{TR}_{\mathrm{TF}}(M)$, it is necessary to evaluate each pair of prefixes of both sequences. In the worst case, there are $n(n-1)k$ requirements. Assuming that γ is the longest common prefix of both α and β, there are $|\alpha| + |\beta| - |\gamma|$ prefixes. In the worst case, we have that $|\gamma| = 0$. Thus, the complexity of determining $\mathrm{TS}_{\mathrm{TF}}(M, \{\alpha, \beta\})$ is $n(n-1)k(|\alpha| + |\beta|)^2$. As there are $t(t-1)/2$ such pairs of sequences $\alpha, \beta \in T$, where t is the number of

Algorithm 1. Greedy Test Minimization Algorithm
Input:
FSM $M = (S, s_0, I, O, D, \lambda, \delta)$;
A set of input sequences T;
A given coverage criterion K;
Output:
$T' \subseteq T$, such that $TS_K(M, T') = TS_K(M, T)$.

```
1.  T' := ∅
2.  R := T
3.  while TS_K(M, T') ≠ TS_K(M, T) do
4.  begin
5.    chosen := ∅
6.    minRatio := ∞
7.    p := 1
8.    while chosen = ∅ do
9.     begin
10.      for each P ⊆ R, such that |P| = p do
11.      begin
12.        cover := TS_K(M, T' ∪ P) \ TS_K(M, T')
13.        if cover ≠ ∅ then
14.        begin
15.          ratio := (w(T' ∪ P) - w(T')) / |cover|
16.          if ratio < minRatio then
17.          begin
18.            chosen := P
19.            minRatio := ratio
20.          end
21.        end
22.      end
23.      p := p + 1
24.    end
25.    R := R \ pref(chosen)
26.    T' := T' ∪ chosen
27.  end
```

Figure 4 *Greedy test minimisation algorithm*

doi: 10.1049/iet-sen.2008.0018

sequences in T, the complexity of determining $TS_{TF}(M, T)$ is of the order $O(t^2n^2kl^2)$, where l is the maximum length of sequences α and β.

Note that if a given test suite contains just one sequence, the algorithm cannot remove it, but can still shorten it if we use the set of prefixes of all tests as the input to the algorithm. Thus, to further minimise a test suite T with multiple tests, we should extend it with the set of all prefixes of its sequences, denoted by pref(T). This extension increases the complexity of the input to the algorithm and all computations. In the worst case, the number of elements to be considered now is of the order of $w(T)$, since |pref(T)| is close to $w(T)$, when the test sequences in T do not share many common prefixes. Note that $|\text{pref}(\{\alpha\})| = w(\alpha)$ and $|\text{pref}(\{\alpha, \beta\})| = w(\alpha) + w(\beta) - w(\gamma)$, where γ is the longest common prefix of both α and β. It is easy to see that $|\text{pref}(T)| \leq w(T)$. If the computational cost of including all the test prefixes is too high, a compromise option may be including a subset of them, preferably with those that are proper prefixes of several tests, since in this way the number of sequences to be considered can be smaller. Note that in the step in Line 25, not only the sequences that are chosen but also all their prefixes are removed from R. This can be done because of the fact that for any test coverage criterion K and any test suite T, if $\alpha \in \text{pref}(T)$, we have that $TS_K(M, T) = TS_K(M, T \cup \{\alpha\})$.

6 Experimental results

In the following sections, we present the settings and results of the experiments that we carried out to answer the questions stated in Section 4.

6.1 Average length against upper bounds

We address here the question: for each criterion, how does the average length of the adequate test suites compare with the upper bounds? The formulae of the upper bounds of the test length (Table 1) contain the major FSM parameter n, the number of states, which is varied in our experiments from 3 to 20. For each value of n, we generate 1000 initially connected deterministic FSMs with four inputs and four outputs for each of the following degrees of completeness: 0.4, 0.6, 0.8, 0.9 and 1.0. Thus, for each value of n, we generate 5000 FSMs, totalling 90 000 FSMs. Fig. 5 shows the maximal test length defined by the corresponding formulae for the upper bounds (for complete and partial FSMs, when applicable), the expected test length obtained with the parameters expected for 'almost all FSMs', and the average length for state, transition, IF and TF coverage criteria. The average length of adequate tests in our experiments is far below the worst-case length. Moreover, we notice that it grows not as fast as the upper bounds suggest. It is, thus, interesting to determine how the average length grows for the various criteria. For SC and IF coverage criteria, we model this growth as a function of the form $f(n) = an^b + c$, where n is the number of states, for some parameters a, b and c. For TC and TF coverage criteria, we model this growth as a function of the form $f(n) = akn^b + c$, where n is the number of states and k is the number of inputs for some constants a, b and c. Note that $k = 4$ for all the FSMs we have generated in this experiment. The forms of these formulae are chosen to resemble the theoretical upper bound formulae. We use the implementation of the nonlinear least-squares (NLLS) Marquardt–Levenberg algorithm [12] available in the 'gnuplot' tool to the values of a, b and c that make $f(n)$ fit best to the collected data. The resulting functions are given in Table 2.

The table also contains the computed ratios of the test length, which allow one to estimate the price of changing a coverage criterion in terms of the increase in the expected test length. As an example, the increase in the test length by switching from an SC criterion to a TC criterion is approximated by the function $0.893kn^{0.08}$. Thus, all things being equal, a TC-adequate test suite is roughly 3.5 times larger than an SC-adequate test suite (recall that $k = 4$ for the FSMs we have considered) while the number of states has a marginal impact. Note that in the ratio between IF-adequate and SC-adequate tests, $2.023n^{-0.09}$, the impact of the number of states is negative, which implies that the difference in the length of test suites for both criteria tends to decrease as the number of states increases. As the number of states increases, so does the accessibility degree of the FSM, and longer sequences more likely contain sequences needed to distinguish the initial state from other states; thus, their impact on the test length diminishes.

6.2 Criteria relative strength

Addressing our third question, 'Given a test suite adequate for one criterion, how adequate is it for another stronger

Table 1 Formulae for the test length for state, transition, IF and TF coverage criteria

Coverage criterion	Maximum length for all FSMs	Expected length for almost all (complete) FSMs
SC	$n(n-1)$	$n \log_k n$
TC	kn^2	$kn \log_k n$
IF	CFSMs: $3(n-1)^2$ PFSMs: $(n+1)(n-1)^2$	$(n-1)(\log_k n + 2 \log_k \log_l n)$
TF	CFSMs: $2kn(2n-1)$ PFSMs: $kn^2(n+1)$	$2kn(1 + \log_k n + \log_k \log_l n)$

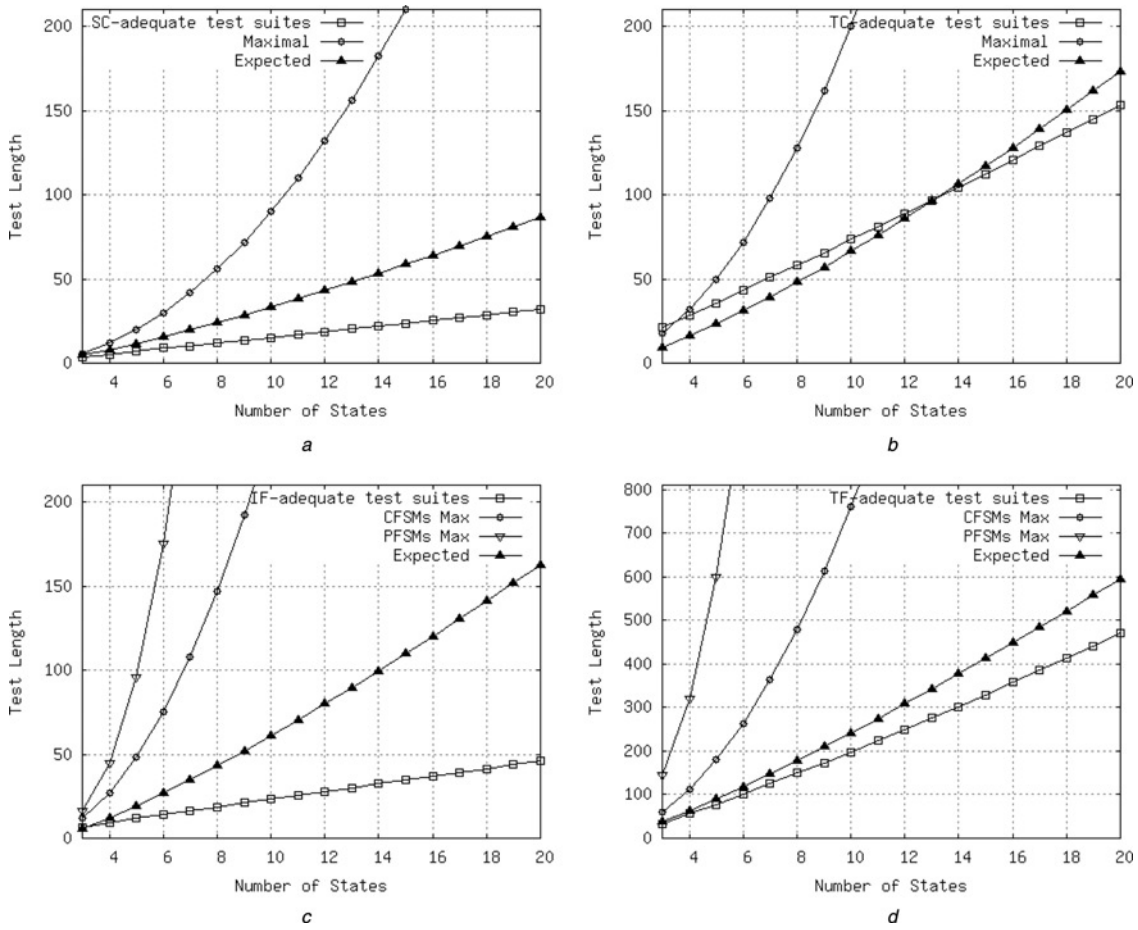


Figure 5 *Maximum, expected and average lengths of adequate test suites with respect to the number of states for:*

a State coverage
b Transition coverage
c Initialisation fault coverage
d Transition fault coverage

criterion?', we determine the coverage of a test suite adequate for one criterion with respect to other criteria. We randomly generate 5000 FSMs with two inputs, two outputs, the number of states ranging from 3 to 20, and the degrees of completeness of 0.4, 0.6, 0.8, 0.9 and 1.0. An adequate test suite (a 'super' test suite) is obtained for each FSM and criterion; then its coverage for the other criteria is determined. For instance, given an SC-adequate test suite, we calculate the percentage of covered transitions. Table 3 shows the relative strength of the four criteria. We present both the average and the standard deviation. For instance, we can observe that a TC-adequate test suite covers on an average 0.928 of the testing requirements of the IF coverage criterion, with the standard deviation of 0.122. Note that, as we generated both reduced and unreduced FSMs, some test suites that are adequate for TF coverage criterion are not adequate even for SC, since there may exist some states that are not distinguishable from any other states. In this case, the TF coverage criterion does not require covering all the states. However, as expected, the test suites adequate for this criterion are almost always adequate for any of the other criteria.

Table 2 Fitted formulae and ratios for the test length for state, transition, IF and TF coverage criteria

Coverage criterion	Fitted formulae	Ratios
SC	$1.31n^{1.07} - 0.23$	–
TC	$1.17kn^{1.15} + 6.31$	$TC/SC = 0.893kn^{0.08}$
IF	$2.65n^{0.96} - 2.25$	$IF/SC = 2.023n^{-0.09}$
TF	$2.17kn^{1.33} + 7.34$	$TF/TC = 1.855n^{0.18}$ $TF/IF = 0.819kn^{0.37}$

doi: 10.1049/iet-sen.2008.0018

Table 3 Relative strength of FSM coverage criteria

	SC	TC	IF	TF
SC	–	1.000/0.000	0.970/0.064	0.994/0.034
TC	0.679/0.132	–	0.772/0.107	0.989/0.047
IF	0.645/0.248	0.928/0.122	–	0.993/0.053
TF	0.299/0.182	0.691/0.134	0.478/0.171	–

6.3 FSM parameters

Addressing the question: 'Which FSM parameters contribute more to test explosion and for which of the four criteria?', we investigate the effect of various FSM parameters on the length of the test suites for the four criteria. We observe that the impact of the number of states is essential, as discussed in Section 4. Here, we are interested in other parameters that characterise an FSM, namely, the number of inputs, outputs and transitions.

Fig. 6*a* shows how the test suite length varies with the number of inputs. We generate FSMs with ten states, two outputs, the number of inputs ranging from two to seven and the degrees of completeness of 0.4, 0.6, 0.8, 0.9 and 1.0 (100 FSMs for each setting, totalling 3000 FSMs). The obtained data indicate that, with respect to the number of inputs, the test length grows almost linearly for transition and TF coverage criteria. At the same time, the number of inputs does not impact the test length for state and IF coverage criteria.

Fig. 6*b* shows how the test length for considered criteria depends on the number of outputs. We generate FSMs with ten states, two inputs, the number of outputs ranging from two to ten and the degrees of completeness of 0.4, 0.6, 0.8, 0.9 and 1.0 (100 FSMs for each setting, totalling 4500 FSMs). We observe that, as expected, the test length for state and TC criteria does not significantly depend on the number of outputs. On the other hand, the length of tests adequate for the TF coverage criterion decreases when the number of outputs increases. The reason is that the length of separating sequences tends to decrease if an FSM has more outputs. Accordingly, the length of test suites for criteria that rely on separating sequences tends to decrease as well. Although the length of a test suite adequate for the IF coverage criterion which also uses separating sequences should also depend on the number of outputs, its impact on the length is negligible in the performed experiments.

Fig. 6*c* shows how the test suite length varies with the number of transitions. Recall that, for fixed numbers of states and inputs, the number of transitions determines the degree of completeness of the FSMs. We generate FSMs with ten states, two outputs, two inputs, and with the number of transitions ranging from 12 to 20 (100 FSMs for each setting, totalling 900 FSMs). We observe that the test length for state and IF coverage criteria does not vary, whereas that for transition and TF coverage criteria grows quasi-linearly.

Dorofeeva *et al.* [5] point out that the length of test suites generated by Wp, HSI, UIOv and H methods is of the order $4n^2$. These methods generate n-complete test suites. In our experiment, we generated SC-TC-IF-TF-adequate test suites, which are also n-complete for reduced FSMs. We expected that the test suite lengths for the SC-TC-IF-TF-adequate test suites were also of the same order. In Fig. 6*d*, we present the average length of SC-TC-IF-TF-adequate test suites and the curve $4n^2$. For each value of n, the average is computed over test suites generated for 900 complete reduced FSMs with four inputs and four outputs, totalling 16 200 FSMs. In the experiments of Dorofeeva *et al.* 1100 complete reduced FSMs are generated with the FSM parameters different from ours, in particular, the number of states ranges from 30 to 100 and the number of inputs and outputs from six to ten. Although the different settings hinder the comparison of obtained data, we observe that our experimental data do not confirm the conclusion of Dorofeeva *et al.* We fitted the data to the $f(n) = a\ n^b + c$ with NLLS and obtained $13.01\ n^{1.418} - 3.697$. The data suggest that the length of n-complete test suites in our experiments grows slower than $O(n^2)$. However, this observation must be checked with more experiments.

6.4 n-completeness of adequate test suites

To address the question, 'How probable is that test suites adequate for various criteria are n-complete?', we determine the percentage of test suites adequate for each criterion which are n-complete. Recall that such tests are guaranteed to deliver the perfect mutation score of 100%, as by definition they kill each and every possible mutant with at most n states. The n-completeness of a test suite is difficult to determine. A negative answer can only be given for small FSMs and in some special cases when exhaustive mutant enumeration is possible. Since this is usually unfeasible, we decided to avoid mutation score calculation and rely on the fact that n-complete test suites can be identified by checking whether known sufficient conditions for n-completeness are satisfied [13]. Thus, we used the algorithm presented in [13] to check whether the test suites

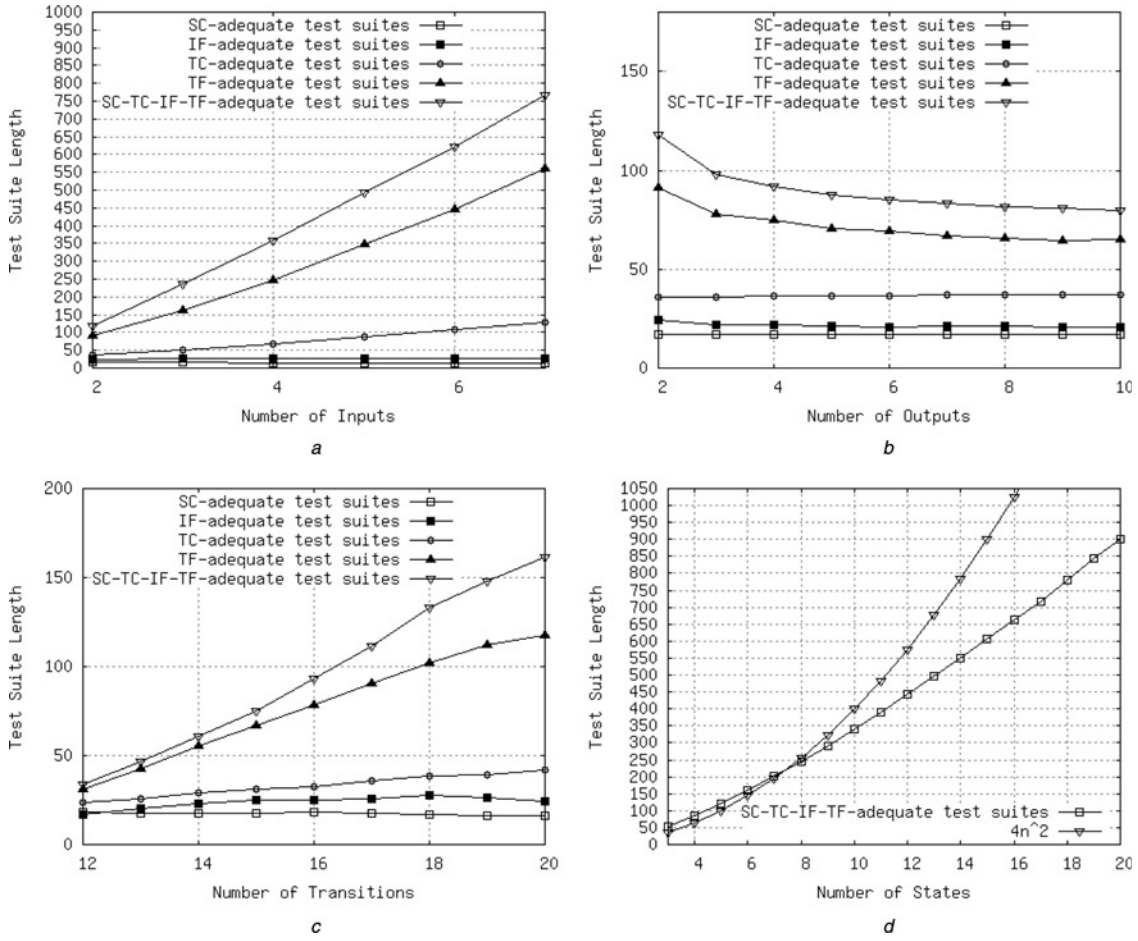


Figure 6 *Average length of adequate test suites for each coverage criteria with respect to*

a Number of inputs
b Number of outputs
c Number of transitions
d Average length of SC-TC-IF-TF-adequate test suites against the curve $4n^2$

are n-complete. We randomly generate 7200 reduced FSMs with two inputs, two outputs, the number of states ranging from 3 to 20, and the degrees of completeness of 0.6, 0.8, 0.9 and 1.0. An SC-TC-IF-TF-adequate test suite is obtained for each FSM and each criterion; and then it is checked whether the test suite satisfies the sufficient conditions for n-completeness. Fig. 7 shows how the percentage of n-complete test suites adequate for each criterion varies as the number of states increases. We observe that the test suites adequate for state and initialisation coverage criteria are n-complete only for FSMs with fewer than five states. Even for those FSMs, the percentage of n-complete test suites is lower than 10%. For the TC criterion, the adequate test suites are n-complete only for FSMs with fewer than nine states. For the TF coverage, the percentage of n-complete adequate test suites is always above zero, decreasing rapidly as the number of states increases.

Note that in this experiment, the FSMs have only four distinct degrees of completeness. To investigate how the degree of completeness of the FSM impacts the probability of obtaining an n-complete test suite adequate for the various criteria, we set up another experiment. We generate 8000 reduced FSMs with two inputs, two outputs, ten

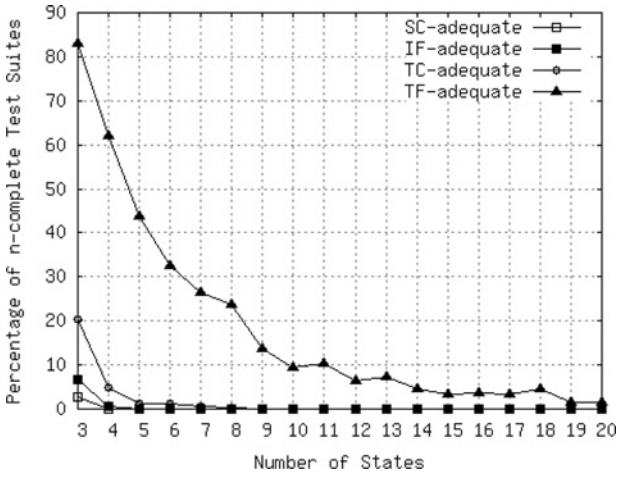


Figure 7 *Variation of the percentage of n-complete test suite adequate for each criterion with respect to the number of states*

doi: 10.1049/iet-sen.2008.0018

states and number of transitions ranging from 13 to 20. An adequate test suite is obtained for each FSM and each criterion. The *n*-completeness of the adequate test suite is then checked. Fig. 8 shows the variation of the percentage of the *n*-complete test suites with respect to the number of transitions. We observe that the percentage of *n*-complete adequate test suites increases as the number of transitions decreases, that is, the less defined the FSM, the more likely a test suite adequate for this criterion is *n*-complete. Considering the TF coverage criterion, we observe that for FSMs with 13 transitions (degree of completeness 0.363), more than 80% of the test suites are *n*-complete. As the number of transitions increases, the percentage of *n*-complete test suites decreases rapidly. When the FSM has 18 states or more (degree of completeness of at least 0.818), the percentage of *n*-complete test suites is lower than 5%.

The above experiments indicate that the chances of obtaining test suites with high fault detection power using less powerful criteria are small. One of possible practical implications is that the actual fault detection power of test suites adequate for the coverage criteria considered in this paper is not high. More precise characterisation of the relative fault detection power of the criteria needs experiments with exhaustive enumeration of complete mutants within a given number of states. On the other hand, the latter can hardly be implemented for FSMs bigger than the ones considered in our experiments.

7 Threats to validity

There are several caveats in interpreting the experimental results, which must be noted:

1. As discussed in Section 5.1, FSMs used in our experiments are randomly generated. As a result, it remains unknown how close they are to 'realistic' FSM specifications. It may be the case that some conclusions drawn based on random state machines do not completely apply to all practical situations. Checking them against state machines adequate to a particular application domain is advised.

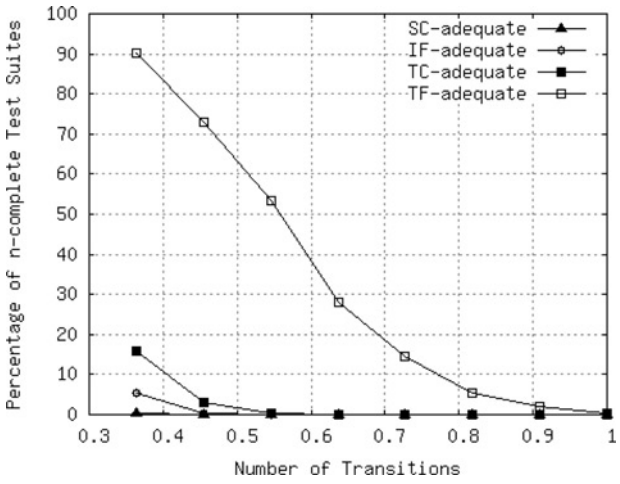


Figure 8 *Variation of the percentage of n-complete test suite adequate for each criterion with respect to the number of transitions*

2. As explained in Section 5.1, to ensure that only initially connected FSMs are generated, initially a tree FSM with the required number of states and the minimal number of transitions is first randomly generated and then more transitions are added. This procedure tends to generate FSMs in which the states with a lower accessibility degree may have more defined transitions than the states with a higher accessibility degree, especially for partial FSMs with a few transitions. As the number of transitions increases, the transitions tend to be more normally distributed. A possible approach that could be used to bypass this problem would be to randomly generate an FSM, and then check whether it is initially connected. However, this approach does not look practical, since the probability of generating an initially connected FSM by a random FSM generator is not high.

3. As previously stated, in order to not bias a test suite by test generation methods, we use a single method to generate test suites that are adequate for all the considered criteria and then minimise them using the same minimisation method, solving a set-cover problem. Another approach that could be tried here is to generate tests using several alternative techniques for obtaining tests adequate for a given criterion and to consider an average test length. For instance, we may generate a TC adequate test suite by determining a transition tour. However, the comparison would still be biased by the methods selected for generation.

4. In the main algorithm for generating tests, for each pair of states, we determine in advance a shortest separating sequence, which is used throughout the algorithm. This approach is similar to traditional test generation methods, such as W, Wp and HSI. However, Dorofeeva *et al.* [14] demonstrate that a shorter test suite can be obtained if the separating sequences are determined on-the-fly. If shorter tests may thus be generated, test suites adequate for the TF and IF coverage criteria may also be shorter than the ones obtained in our experiments. Even if the charts for the test length may further be refined, the obtained characterisation of adequate tests and their ratios may well persist.

5. The test minimisation is a computationally hard problem. Hence, approximation algorithms based on greedy approaches have been employed; as a result, the minimised test suites are not guaranteed to be minimal. The replication of these experiments with another minimisation algorithm may allow to factor out the impact of a minimisation algorithm on the adequate test length.

6. We use sufficient conditions for a check of *n*-completeness. Therefore there may exist *n*-complete test suites that

violate them. Thus, our experiments report conservative values for the n-completeness probability, since the actual value may be even higher.

8 Conclusion

This paper is devoted to experiments with common coverage criteria used for FSM-based testing. We developed an approach for generating tests adequate for each of the criteria in such a way that the results do not significantly depend on methods used for test generation from randomly generated FSMs. The idea is to first generate a test suite that is adequate for all the considered criteria and then minimise it for each criterion separately using a single test minimisation algorithm that solves a combinatorial set-cover problem. We proposed a generalised greedy algorithm, which is used to minimise test suites with respect to a given coverage criteria. The algorithm implements a heuristics based on the idea that pairs of sequences should be considered only when single sequences cannot cover a new testing requirement. The experiments showed that the proposed algorithm is faster than algorithms found in the literature, at the cost of a small loss of reduction power. The prototype tool environment developed for the experiments has a much wider application area, as it can be used to actually generate tests adequate for various test coverage criteria. It can treat not only criteria that may require pairs of sequences to cover some testing requirements, but also more complex criteria which may require a larger number of sequences.

The obtained experimental data shed some light on the expected length of test suites adequate for state, transition, initialisation and TF coverage criteria. In particular, the experiments show that, as expected, the tests are much shorter than the upper limits suggest. Moreover, the average length of test suites grows much slower than the corresponding formulae suggest. For instance, the length of test suites adequate for the TF coverage criterion are of the order $O(kn^{1.33})$, which is lower than the theoretical $O(kn^3)$. The formulae for the expected length of the test suites for the four criteria, which we derived using some known (although rarely used) results on the asymptotic characterisation of FSMs, give values much closer to experimental data than worst-case estimations. We have also compared the relative strength of the criteria. As the TF coverage criterion subsumes TC and IF coverage criteria only for reduced FSMs, the experimental results suggest that, even for unreduced ones, test suites adequate for the TF coverage criterion, cover, on average, about 99% of the requirements of TC and IF coverage criteria.

The experiments confirmed that the number of states has the greatest impact on the length of the test suites adequate for all criteria. The number of inputs influences almost linearly the length for TC and TF coverage criteria. At the same time, the number of inputs does not impact the test length for state and initialisation coverage criteria. An increase in the number of outputs does not lead to an increase in the test length for SC and TC criteria. On the other hand, the test length for IF and TF coverage criteria tends to decrease with the growth in the number of outputs, because of the resulting shortening of separating sequences. As expected, the number of transitions has a nearly linear impact on the test length for transition and TF coverage criteria, with no sensible influence on the length of tests adequate for SC and IF coverage criteria. Our experimental data also suggest that the length of n-complete test suites increases slower than $O(n^2)$, as concluded in a previous work. However, as the parameters of the FSMs generated in the experiments differ, more experiments are necessary to draw a more definitive conclusion. Our experiments also indicate that test suites adequate for TF coverage criterion have a fairly high probability of being n-complete for small FSMs. Moreover, they demonstrate that the chances of obtaining test suites with high fault-detection power are small for test suites adequate for the coverage criteria considered in this paper.

We need to conduct more experiments also to refine the formula to estimate the test length that we suggested. In our fitted formulae, we only allow the variation of the number of states, using a fixed number of inputs. It would be interesting to find fitted formulae that include both variables. We also intend to assess the variation in the test suite length with respect to other FSM parameters, such the accessibility degree, distinguishability degree and distinguishability ratio. It would be interesting to try to enrich the experimental data using more realistic data obtained with the help of testers, for example, along with random generation of FSMs and test suites, one could consider FSMs and test suites manually built by testers. Finally, it would also be interesting to investigate how FSM coverage criteria relate to those of the program code, which implements state machines.

9 Acknowledgments

The authors would like to thank FAPESP, CNPq and NSERC for their partial financial support of this work, as well as Prof. Mario de Castro Andrade Filho for his help in the statistical analysis. The reviewers are acknowledged for their useful comments.

10 References

[1] BOCHMANN G.V., PETRENKO A.: 'Protocol testing: review of methods and relevance for software testing'. ACM Int. Symp. Software Testing and Analysis (ISSTA'94), USA, 1994

[2] BINDER R.: 'Testing object-oriented systems' (Addison-Wesley Inc., 2000)

doi: 10.1049/iet-sen.2008.0018

[3] FRANKL P.R., WEYUKER E.J.: 'A formal analysis of the fault-detecting ability of testing methods', *IEEE Trans. Softw. Eng.*, 1993, **19**, (3), pp. 202–213

[4] MOORE E.F.: 'Gedanken-experiments on sequential machines', 'Automata studies, Annals of mathematics series' (Princeton University Press, Princeton, New Jersey, 1956), vol. 34, pp. 129–153

[5] DOROFEEVA R., YEVTUSHENKO N., EL-FAKIH K., CAVALLI A.R.: 'Experimental evaluation of fsm-based testing methods'. 3rd IEEE Int. Conf. Software Engineering and Formal Methods (SEFM 2005), 2005, pp. 23–32

[6] PETRENKO A., BOCHMANN G.V., YAO M.: 'On fault coverage of tests for finite state specifications', *Comput. Netw. ISDN Syst.*, 1996, **29**, (1), pp. 81–106

[7] CHOW T.S.: 'Testing software design modeled by finite-state machines', *IEEE Trans. Softw. Eng.*, 1978, **4**, (3), pp. 178–187

[8] TRAKHTENBROT B.A., BARZDIN Y.M.: 'Finite automata, behaviour and synthesis' (North-Holland Pub. Co., 1973)

[9] YEVTUSHENKO N., PETRENKO A.: 'Synthesis of test experiments in some classes of automata', *Automat. Control Comput. Sci.*, 1990, **24**, (4), pp. 50–55

[10] KARP R.M.: 'Reducibility among combinatorial problems', in MILLER R.E., THATCHER J.W. (EDS.): 'Complexity of computer computations' (Plenum Press, 1972), pp. 85–103

[11] HASSIN R., SEGEV D.: 'The set cover with pairs problem'. Proc. 25th Annual Conf. Foundations Software Technology and Theoretical Computer Science, 2005, pp. 164–176

[12] BATES D.M., WATTS D.G.: 'Nonlinear regression and its applications' (Wiley, 1988)

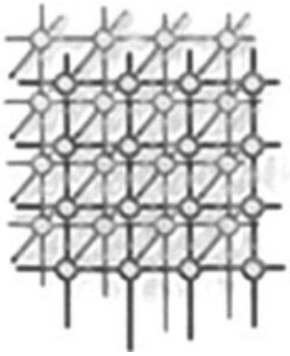
[13] SIMÃO A., PETRENKO A.: 'Checking FSM test completeness based on sufficient conditions'. CRIM-07/10-20, Montreal, Quebec, Canada, 2007

[14] DOROFEEVA R., EL-FAKIH K., YEVTUSHENKO N.: 'An improved conformance testing method'. Formal Techniques for Networked and Distributed Systems, 2005, (*LNCS*, **3731**), pp. 204–218

Apêndice F

S. R. S. Souza, S. R. Vergílio, P. S. L. Souza, A. S. Simão, A. Hausen. Structural Testing Criteria for Message-Passing Parallel Programs. Concurrency and Computation. Practice & Experience, v. 20, p. 1893-1916, 2008.

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

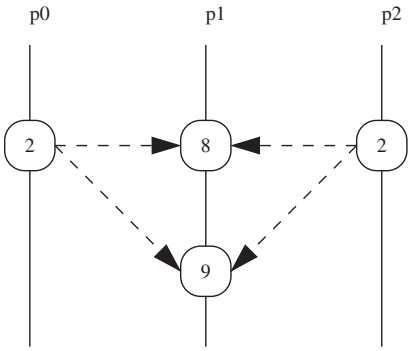


Figure 1. Example of non-determinism.

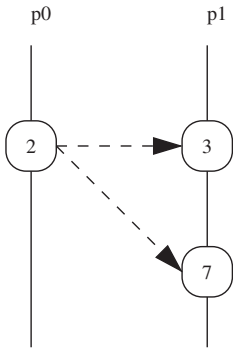


Figure 2. Example of non-blocking receive.

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

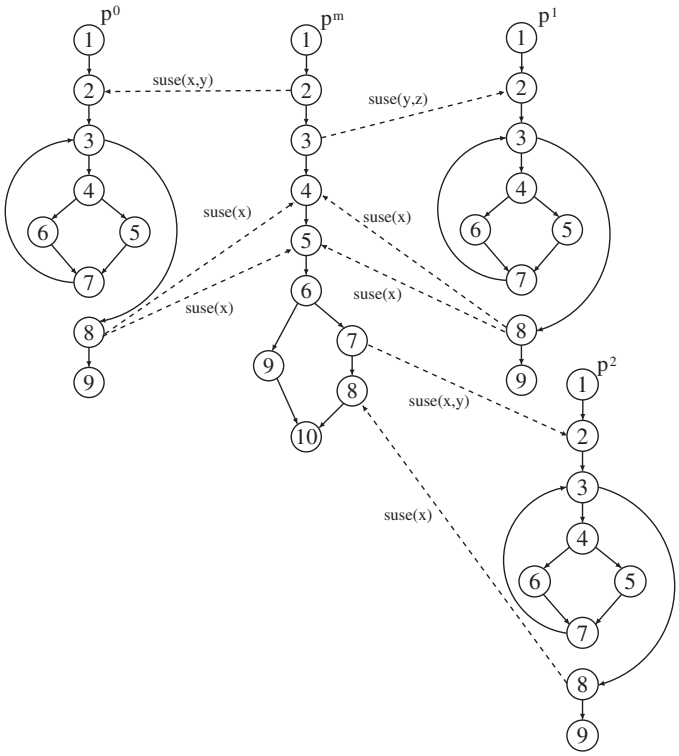


Figure 4. Parallel control-flow graph for GCD program.

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

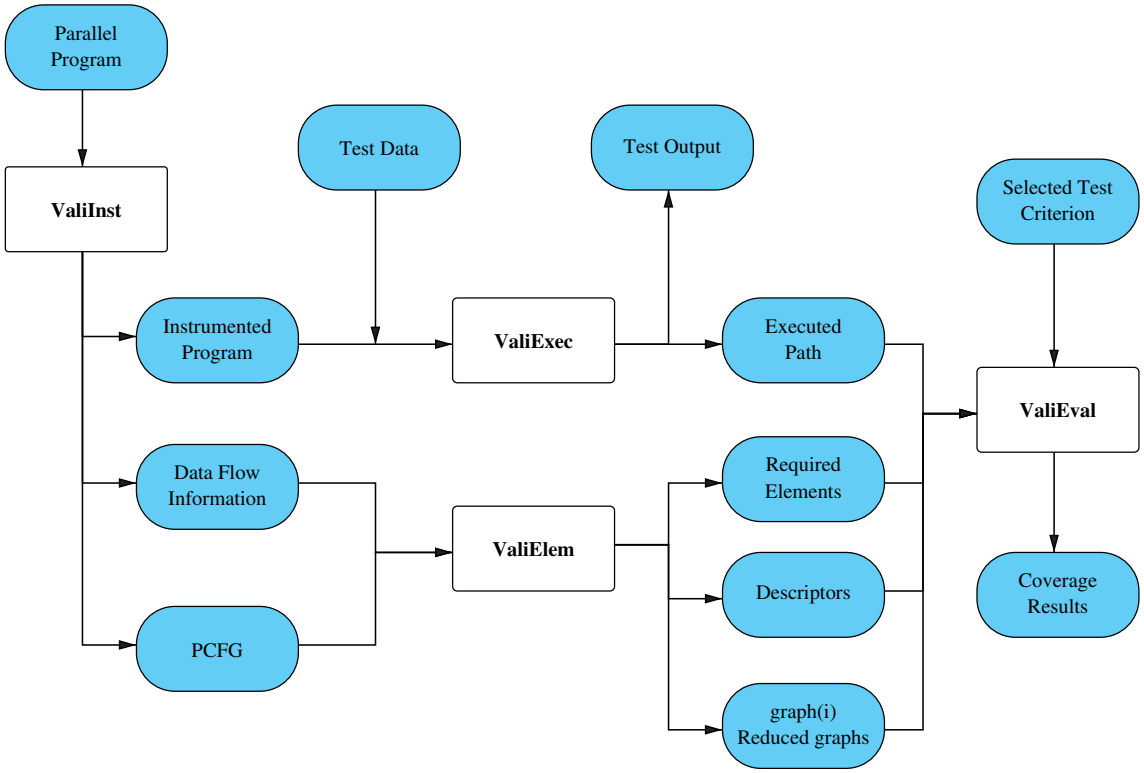


Figure 5. ValiPar tool architecture.

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 G

A. S. Simão, J. C. Maldonado and R. S. Bigonha. A Transformational Language for Mutant Description. Computer Languages, Systems & Structures, v. 35, p. 322-339, 2009.

Computer Languages, Systems & Structures 35 (2009) 322 – 339

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

A transformational language for mutant description

Adenilso Simão[a,*], José Carlos Maldonado[a], Roberto da Silva Bigonha[b]

[a]Departamento de Sistemas de Computação, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, Av. Trabalhador São-carlense, 400 - Centro, P.O. Box: 668, ZIP: 13560-970 São Carlos, SP, Brazil
[b]Departamento de Computação, Universidade Federal de Minas Gerais, Brazil

ARTICLE INFO

Article history:
Received 9 December 2006
Accepted 29 October 2008

Keywords:
Mutation testing
Transformation languages
Logical languages
Software testing
Prototyping

ABSTRACT

Mutation testing has been used to assess the quality of test case suites by analyzing the ability in distinguishing the artifact under testing from a set of alternative artifacts, the so-called mutants. The mutants are generated from the artifact under testing by applying a set of mutant operators, which produce artifacts with simple syntactical differences. The mutant operators are usually based on typical errors that occur during the software development and can be related to a fault model. In this paper, we propose a language—named $\mathscr{MuDeL}$ (MUtant DEfinition Language)—for the definition of mutant operators, aiming not only at automating the mutant generation, but also at providing precision and formality to the operator definition. The proposed language is based on concepts from transformational and logical programming paradigms, as well as from context-free grammar theory. Denotational semantics formal framework is employed to define the semantics of the $\mathscr{MuDeL}$ language. We also describe a system—named `mudelgen`—developed to support the use of this language. An executable representation of the denotational semantics of the language is used to check the correctness of the implementation of `mudelgen`. At the very end, a mutant generator module is produced, which can be incorporated into a specific mutant tool/environment.

1. Introduction

Originally, mutation testing [1,2] is a testing approach to assess the quality of a test case suite in revealing some specific classes of faults, and can be classified as a fault-based testing technique. Although it was originally proposed for program testing [2], several researchers have applied its underlying concepts in a variety of other contexts, testing different kinds of artifacts, e.g., specifications [3–7], protocols testing [8] and network security models [9]. Moreover, mutation testing has been employed as a useful mechanism to improve statistical validity when testing criteria are compared, such as in [10].

The main idea behind mutation testing is to use a set of alternative artifacts (the so-called *mutants*) of the artifact under testing (the *original* artifact) to evaluate test case sets. These mutants are generated from the original artifact by introducing simple syntactical changes and, thus, inducing specific faults. Usually, only a simple modification is made in the original artifact. The resulting mutants are the so-called 1-*order* mutants [11]. A *k-order* mutant can be thought of as a mutant in which several 1-*order* mutations were applied [12]. The ability of a test case suite in revealing these faults is checked by running the mutants and comparing their results against the result of the original artifact for the same test cases.

The faults considered to generate the mutants are based upon knowledge about errors that typically occur during software development and can be associated to a fault model. In the mutation testing approach, the fault model is embedded in the *mutant operators* [13]. A mutant operator can be thought of as a function that takes an artifact as input and produces a set of mutants,

* Corresponding author. Tel.: +55 16 3373 9700; fax: +55 16 3371 2238.
E-mail addresses: adenilso@icmc.usp.br (A. Simão), bigonha@dc.ufmg.br (R. da Silva Bigonha).

1477-8424/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2008.10.001

in which the fault modeled by that particular operator is injected. The fault model has great impact in the mutation testing cost and effectiveness, and, hence, so does the mutant operator set. In general, when the mutation testing is proposed for a particular artifact, one of the first steps is to describe the fault model and a mutant operator set.

Considering the important role of mutant operators to the mutation testing, their definition and implementation are basic issues for its efficient and effective application. The mutant operator set has to be assessed and evolved to improve its accuracy w.r.t. the language in question. This is usually made by theoretical and/or empirical analysis. Specifically for empirical analysis, it is necessary to design and construct a prototype or a supporting tool, because manual mutant generation is very costly and error-prone. However, the tool design and construction are also costly and time-consuming tasks. An approach that can be used to tackle this problem is to establish prototyping mechanisms that provide a low-cost alternative, making easier the evaluation and evolution of the mutant operators without requiring too much effort to be expended in developing tools. At the very end, the produced mutant generator module may be incorporated into a specific mutant tool/environment.

Another important issue to be considered is that, given the already mentioned impact on the mutation testing effectiveness, mutant operators must be described in a way as rigorous as possible, in order to avoid ambiguities and inconsistencies. This is similar to what happens to other artifacts of software engineering. Several initiatives towards defining mutant operators for different programming languages can be found in the literature [14–19]. Although we can identify some approaches in which the operators are formally defined (e.g., [20]), in most of the cases, the definition is informal and based on a textual description of the changes that are required in order to generate the mutants (see, e.g., [14]).

From these works, we can observe that there are common conceptual mutations amongst different languages, such as Fortran, C, C++, Statecharts, FSMs and so on, although this point has not been explicitly explored by the authors. This fact motivated us to investigate mechanisms to design and validate mutation operators as independent as possible of the target language. This same scenario leads to opportunities to reuse the knowledge underlying the mutations (i.e., effectiveness, costs related to the generation of mutants, to determination of equivalent mutants, to the number of test cases required to obtain an adequate test case set) of particular mutations, and of the related operators [21,22].

In this paper, we present a language—called $\mathcal{MuDeL}$ (MUtant DEfinition Language)—for the definition of mutant operators, a tool to support the language and case studies that show how these mechanisms have been employed in several different contexts. The language was designed with concepts from transformational [23] and logical [24] paradigms. Its motivation is threefold. Firstly, $\mathcal{MuDeL}$ provides a way to precisely and unambiguously describe the operators. In this respect, $\mathcal{MuDeL}$ is an alternative for sharing mutant descriptions. We employed *denotational semantics* [25,26] to formally define the semantics of $\mathcal{MuDeL}$ language [27]. Observe that the description of the mutant are syntax driven and the semantics of the mutant itself are not taken into consideration. Secondly, the mutant operator description can be "compiled" into an actual mutant operator, enabling the mutant operator designer to validate the description and potentially to improve it. With this purpose, we have implemented the `mudelgen` system. Given a mutant operator defined in $\mathcal{MuDeL}$ and the original artifact, the `mudelgen` compiles the definition and generates the mutants, based on a given context-free grammar of the original artifact. The denotational semantics of $\mathcal{MuDeL}$ was used as a pseudo-oracle (in the sense discussed by Weyuker [28]) in the validation of the `mudelgen` [27]. And finally, by providing an abstract view of the mutations, $\mathcal{MuDeL}$ eases the reuse of mutant operators defined for syntactically similar languages. For example, although the actual grammars of, say, C and Java are quite different, they both share several similar constructions, and, by carefully designing their grammars and the mutant operators, one can reuse the mutant operators that operate on the same construction, e.g., deleting statements, swapping expressions, and so on, on both languages. We have applied $\mathcal{MuDeL}$ and `mudelgen` with the languages C, C++ and Java and with the specification languages FSMs and CPNs. In particular, we used them in the context of Plavis project, which involves Brazilian National Space Agency. We observe that for languages with similar grammar, we could reuse not only the conceptual framework behind the mutation, but also the $\mathcal{MuDeL}$ mutant operators themselves.

Mutation testing demands several functionalities other than just generating mutants, e.g., test cases handling, mutant execution and output checking. Both $\mathcal{MuDeL}$ and `mudelgen` are to be used as a piece in a complete mutation tool, either in a tool specifically tailored to a particular language or in a generic tool—a tool that could be used to support mutation testing application having the most used languages as target languages. In fact, $\mathcal{MuDeL}$ and `mudelgen` are steps towards the implementation of such generic tools.

This paper is organized as follows. In Section 2 we discuss some related work and summarize the main features of mutant operators, highlighting the requirements for a description language for this specific domain. In Section 3 we present the $\mathcal{MuDeL}$ language and illustrate its main features. In Section 4 we show results from the application of the language we have made up to now, emphasizing the cases where we could effectively reuse mutant operator descriptions in different languages. In Section 5 we discuss relevant implementation aspects of the `mudelgen` system and depict its overall architecture. Finally, in Section 6 we make concluding remarks and point to further work.

2. Mutant operators

Mutation testing has been applied in several context, for several different languages. Therefore, mutant operators have been defined for those applications. The definitions are usually made in an *ad hoc* way, ranging from textual descriptions to formal

definitions. Notwithstanding, to the best of our knowledge, $\mathcal{MuDeL}$ is the first proposal to provide a precise language to describe mutant operators.

Mutation testing was first applied for the FORTRAN language [15]. DeMillo designed 22 mutant operators and developed the MOTHRA tool. The mutant operator descriptions were textual and heavily based on examples. Although the examples are very useful to illustrate the mutant operator, describing it by these means is ambiguous, and does not promote reuse.

Agrawal [14] proposed 77 mutant operators for the C language. The definition were based on the FORTRAN mutant operators. Most C mutant operators are basically a translation of the respective FORTRAN mutant operators. However, since the C language has a much richer set of arithmetic and logical operators, there are more mutant operators for the C language. These operators were implemented in the Proteum tool [29]. (Actually the mutant operators implemented by Delamaro et al. [29] are adapted versions of the Agrawal's ones.) Afterwards, Delamaro et al. [18] proposed mutant operators for testing the interfaces between modules in the C language, named interface mutation. The Proteum tool was extended with these operators, deriving the Proteum/IM [18].

Fabbri [30] investigated mutation testing concepts in the context of specification techniques for reactive systems. Mutant operators were designed for Petri nets, Statecharts and finite state machines (FSMs). Differently from the above approaches, those mutant operators were formally defined, using the same formalism of the corresponding technique.

Kim et al. [16] have proposed a technique named "hazard and operability studies" (HAZOP) [31] to systematically derive mutant operators. The technique is based on two main concepts. It first identifies in the grammar of the target language where mutation may occur and then defines the mutations guided by "Guide Words". They applied their technique to the Java language. Although the resulting operators do not significantly differ from previous works, the proposed methodology is an important step towards a more rigorous discipline in the definition of mutant operators.

From these examples, we can summarize the characteristics of mutant operators used in different context. Usually, from a single original artifact, a mutant operator will generate several mutants. For example, a mutant operator that exchanges a **while** statement into a **do–while** statement will generate as many mutants as the number of **while** statements in the artifact. In each mutant, a single **while** will be replaced by a **do–while**.

The number of mutants that can be generated from a particular artifact is very large. Considering an original artifact, any other artifact in the same language could be considered as a mutant, due to the informal and broad definition of "syntactical change" necessary to generate a mutant. To keep the number of mutants at a tractable level, only mutants with simple changes are considered. Roughly speaking, a change is considered simple when it cannot be decomposed into smaller, simpler changes. For that reason, to describe a mutant operator, usually only one change should be defined.

An important point that should be highlighted is that a change being simple does not mean it is straightforward. The syntax of the artifact should be taken into account, in order to generate syntactically valid mutants. Concerning this point, a mechanism based on simple text replacement is not enough. It is necessary to embody some mechanisms to guarantee that the mutants are also valid artifacts in the original language. The pattern replacement, which is typical in transformational languages, is more suitable in this context.

Sometimes the single logical change implies in changing more than one place in the artifact. For example, a mutant operator for exchanging two constants must indicate that two different but related changes, one in each place where the exchanging constants appear. Moreover, in some cases, although being treated as a single entity, the mutant operator involves different changes in different places. Therefore, it is necessary to be able to relate these different changes into the mutant operator.

Mutations can be classified into two major groups: context-free mutations and context-sensitive ones. Context-free mutations are those that can be carried out regardless the syntactical context in which the mutated part occurs. Conversely, context-sensitive mutations depend upon the context, e.g., the variables visible in a specific scope. Most mutant operators in literature [3,6,14,17] involve context-free mutations. Even for a context-sensitive language, there can be context-free mutations. An example of context-free mutations is the change of "`x = 1`" by "`x += 1`", since wherever the first expression is valid, so is the second. However, the change of "`x = 1`" by "`y = 1`" is context-sensitive, since the second expression will not be valid unless `y` has the same declaration status as does `x`. To tackle this difficult problem, a language for describing mutants should either embody features to specify context-sensitive grammar or provide some way to gather information from the context in some kind of lookup table.

3. $\mathcal{MuDeL}$ language

Based on the characteristics of mutants, we designed a language to allow the definition of mutants in a way as easy, language-independent and natural as possible. However, due to pragmatical issues, we have taken some design decisions that trades off between the goals listed above and the possibility of implementing of an efficient supporting tool. Therefore, $\mathcal{MuDeL}$ does not provide a completely language-neutral mechanism for describing and implementing mutant operator. Indeed, the syntax of the target language should be somewhat embodied in the mutant definition. $\mathcal{MuDeL}$ language is, thus, a mixed language that brings together concepts of both the transformational and the logical paradigms. From the transformational paradigm it employs the concept of pattern matching and replacement. The transformational language that is closest to $\mathcal{MuDeL}$ is TXL [32,33]. TXL has both matching and replacement operations. However, TXL works in a one-to-one basis and has a imperative-like control flow, making unnatural to describe mutant operators. Instead, the control flow of $\mathcal{MuDeL}$ is inspired in Prolog's. The most important similarity of $\mathcal{MuDeL}$ and Prolog's is, however, the way a mutant operator definition is interpreted. Like a Prolog clause,

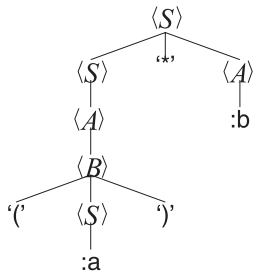


Fig. 1. The pattern tree for ‘`( :a ) * :b`’. The types of ‘`:a`’ and ‘`:b`’ have been declared as ⟨*A*⟩ and ⟨*B*⟩, respectively.

a $\mathcal{MuDeL}$ definition can be thought of as a predicate. A mutant should satisfy the “predicate” of a mutant operator definition in order to be a mutant of this operator w.r.t. the respective original artifact. However, like a `findall` predicate in Prolog, the $\mathcal{MuDeL}$ definition can be used to enumerate all mutants that satisfy it. This is made by the `mudelgen` system, described in Section 5.

3.1. Basic notations

In order to be able to handle different kinds of artifacts, we should choose an intermediate format to which every artifact can be mapped. Assuming that most artifacts can be thought of as elements of a language defined by a context-free grammar, the use of syntax tree has an immediate appeal [34]. Therefore, in the $\mathcal{MuDeL}$ language, every artifact, either a program or a specification, is mapped into a syntax tree. The mapping is carried out by parsing the artifact based on a context-free grammar of the language. The syntax tree can be handled and modified in order to represent the mutations. It is thus necessary a way to describe how the syntax tree must be handled.

We define a set $\mathcal{M}$ of *meta-variables*[1] and extend the syntax tree to allow for leaves to be meta-variables as well as terminal symbols. Moreover, in this extension, the root node can be any non-terminal symbol (not only the initial one, as in the syntax trees). We call these extended syntax trees *pattern trees*, or, if it is unambiguous from the context, just *patterns*. Each meta-variable has an associated non-terminal symbol, which is called its *type*. A meta-variable can be either free or bound. Every bound meta-variable is associated to a sub tree that can be generated from its type. Therefore, a syntax tree is just a special kind of pattern tree; a kind where every meta-variable (if any) is bound. Fig. 1 shows an example of a pattern tree. As a way to distinguish from ordinary identifiers, we prefix the meta-variables with a colon (:). Even in the presence of meta-variables, the children of a node must be in accordance with its artifact, i.e., a meta-variable can only occur where a non-terminal of its type also could.

To specify patterns we use the following notation. The simplest pattern is formed by an anonymous meta-variable, as its root node. This pattern is expressed just by the non-terminal symbol that is its root node enclosed in squared brackets. For example, `[A]` is a pattern whose root node is an anonymous meta-variable of type ⟨*A*⟩. In most cases, such a simple notation will not be enough to specify pattern trees. One can use a more elaborated pattern notation, instead. The non-terminal root symbol is placed in squared brackets, as before, but following it, in angle brackets, a sequence of terminal symbols and meta-variables is included. For example, the pattern tree in Fig. 1 is denoted by `[S<( :a )* :b>]`. Note that inside the angle brackets the grammar of the artifact, rather than the $\mathcal{MuDeL}$'s grammar, is to be respected. Nonetheless, meta-variables come from $\mathcal{MuDeL}$ itself and, thus, the previous pattern will only be valid if the meta-variables :a and :b are declared with proper types. Therefore, instead of being just a language, $\mathcal{MuDeL}$ is indeed a *meta-language*, in that a $\mathcal{MuDeL}$'s definition is valid or not w.r.t. a given source grammar. In other words, given a source grammar of an artifact language, we can instantiate $\mathcal{MuDeL}$ language for that grammar. The source grammar determines the form and the syntax of the pattern trees.

The unification of a tree and a pattern is in the kernel of any transformational system. In the unification, two pattern trees *c* and *m* are taken and an attempt to unify them is done. The unification can either fail or succeed. In case of success, the meta-variables in the pattern trees are accordingly bound to respective tree nodes, in a way that makes them unrestrictly interchangeable. In case of failure, no meta-variable binding occurs. The unification algorithm is similar to Prolog's one [24]. Fig. 2 shows an example of a successful unification. The dashed line indicates the meta-variable bindings.

3.2. Operator structure

A mutant operator definition has three main parts: operator name, meta-variable declarations and body. The operator name declaration comes first. This name is just for documentation purposes and has no impact in the remaining parts of the definition. Next, there is the optional section of meta-variable declarations. If present, this section is started by the keyword `var` followed by a list of one or more meta-variable declarations. A meta-variable declaration is a meta-variable name followed by a pattern tree, which is its type. The last section, enclosed by the keywords `begin` and `end operator`, is the body of the operator, which

[1] We chose the term *meta-variable* instead of the term *variable*, which has a particular meaning in most language to which $\mathcal{MuDeL}$ can be applied.

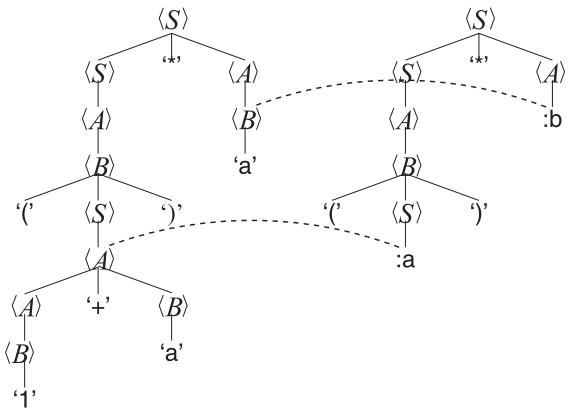


Fig. 2. An example of unification.

```
operator STDL
  var :s [statement]
begin
  * replace [statement<:s>]
    by      [statement<;>]
end operator
```

Fig. 3. A simple mutant operator. For every statement in the program, a mutant is generated by "deleting" the statement.

is a compound mutation operation (explained later). This operation will be executed w.r.t. the syntax tree of the original artifact. Fig. 3 presents a mutant operator definition, illustrating its overall structure. This mutant operator, whose name is STDL, declares the meta-variable :s with the type ⟨*statement*⟩, and has a simple operation as its body, that, as will be clarified later, generates mutants replacing nodes with type ⟨*statement*⟩ by a semi-colon (the null statement), according to the grammar of the C programming language. Observe that there is no explicit indication of which node should be considered by the replace operation, which, in this case, implies that the whole tree should be used.

The body of a mutant definition written in $\mathscr{MuDeL}$ is composed by a combination of *operations*. The syntax of $\mathscr{MuDeL}$'s body part can be divided into operations, combiners and modifiers. An operation can be thought of as a predicate that either modify the syntax tree or control the way the remain operations act. The operations can be joined by combiners. The behavior of an operation can be altered by modifiers.

If a set of operations must be used in several different points in a mutant operator, it is possible to declare a *rule* with these operations and invoke the rule wherever necessary. Rules can be thought of as procedures of conventional programming languages. In this way, mutant operators can be defined in a modular way. Rules can be defined in a separated file and imported in the mutant operator, allowing to reuse similar operations among a set of related mutant operators.

3.3. Operations

An operation is a particular statement about how to proceed in the generation of a particular mutant. An operation takes place in a particular state, which is formed by the current syntax tree. Every operation, being it simple or compound, can result in zero or more alternative syntax trees. If it results in zero alternative syntax tree, we say that the operation result in failure, i.e., that it fails.

3.3.1. Replace operation

The replace operation is the most important one in $\mathscr{MuDeL}$ language, since it is responsible for altering the original syntax tree into the mutated one. It requires three arguments: the tree *c* to be altered, the pattern tree *r*, that is to be unified with *c*, and the pattern tree *b*, that will replace *c* in the case of a successful unification of *c* and *r*. Both *r* and *b* can contain meta-variables that allow to use parts of *c* in the replacement.

The syntax of a replace operation is

```
c @@ replace r by b
```

where *c* must be a meta-variable.

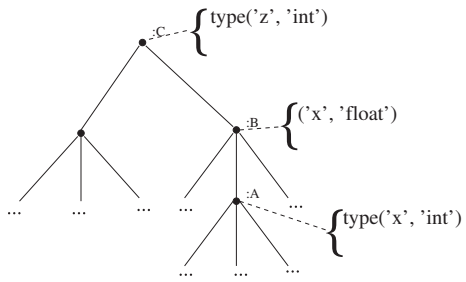


Fig. 4. Facts in the syntax tree.

3.3.2. Match operation

The matching operation can be used to select where the replacement is applied. The matching operation takes two arguments: a tree *c* and a pattern tree *m*. It tries to unify *c*, which must be a meta-variable, and *m*, binding meta-variables in *m* if necessary. The bindings are still active after the unification, allowing to select parts of *c* to be further handled.

3.4. Assert and consult operations

Most of the mutations can be made in a context-free basis. This is true due to fact that if the original artifact is syntactically valid, the mutant operator can safely rearrange some part of it and ensure the mutant is also syntactically correct. However, there are some mutant operators that require some information that comes from the context in which the mutated parts are located. For instance, when exchanging a variable by another one, it is necessary to check whether their declared types are compatible. Strictly speaking, this could be made with the operations, modifiers and combiner described. Nonetheless, such a way of definition will be quite awkward. We tackle this problem by enriching the syntax tree with *attributes* [35]. The attributes are a set of tuples that has a name and a set of values and is associated to a node in the syntax tree. The attributes are calculated and stored when the syntax tree is built. (See Section 5 for a discussion on how the attributes are calculated.)

To access the values of the attributes, $\mathcal{MuDeL}$ has the `consult` operation. The consult operation takes an starting tree node *c*, an attribute name *n*, and a list of meta-variables or pattern trees, which represent the arguments of the attribute. The operation will look for any tuple with name *n* in the tree node *c*. If it finds any, it will try to unify the list of arguments with the list of arguments in the tuple. Each tuple that successfully matches the list of arguments will produce an alternative state. However, when there is no tuple in *c* with name *n*, the consult operation will recursively search in the parent node of *c*, until a node with such a tuple is found or it has already searched in the root of the tree (that has no parent). Observe that this upward search embodies the way context information is usually dealt with. It also allows the correct dealing with a scope of most typed languages, in which the attributes of an entity can be overridden in an inner scope. The `consult` operation can be compared to the Prolog `consult` predicate. However, the Prolog's `consult` predicate uses a single global base of facts, while in $\mathcal{MuDeL}$ the facts are scattered over the tree and are searched in a hierarchical way. The operation can be negated, analogously to the matching operation. Fig. 4 illustrates how attributes are stored and retrieved in syntax trees. We annotate in each which tuples are defined. If the operation

```
:C @@ consult type with :v :t
```

is executed with `:v` unified to 'x', it will fail, since there is no fact about 'x' in `:C`. However, if the same operation is executed with `:v` unified to 'z', it will succeed with the unification of `:t` to 'int'. If the operation

```
:B @@ consult type with :v :t
```

is executed with `:v` unified to 'x', it will succeed with the unification of `:t` to 'float'. If the same operation is executed with `:v` unified to 'z', it will succeed with the unification of `:t` to 'int', since this fact is stored in the parent of `:B`. If the operation

```
:A @@ consult type with :v :t
```

is executed with `:v` unified to 'x', it will succeed with the unification of `:t` to 'int'. Observer that the facts in `:B` were overridden.

In most of the cases, the attributes that are consulted are stored in the tree when it is built (see Section 5). However, sometimes, it may be useful to also be able to store tuples in the tree. This is made with the `assert` operation. It takes a context tree *c*, an attribute name *n*, a list of meta-variables or patterns, which represents the arguments of the tuple, and a list of patterns that represent where the tuple should be stored. The operation will search upwards from *c* for a tree node that matches any of the patterns. If it finds such a node, the tuple is stored in it. Otherwise, the tuple will be stored in the root node.

3.4.1. Donothing, abort and cut operations

We include some atomic operations that enhance the control of generation of the mutants. They control the set of alternative states the next operations will deal with. The `donothing` operation, as its name suggests, does nothing at all. It succeed exactly once. It can be thought of as a placeholder for situations where an operation is necessary but no effect is indeed required. It is similar to the `true` predicate of Prolog.

```
:p @@ replace [S< while( :e ) :s >]
      by      [S< do :s while ( :e ); >]
```

Fig. 5. Replace operator.

a

```
while (a < b) {
  if (a % 2 == 0) {
    a ++;
  } else {
    a += 3;
  }
}
```

b

```
do {
  if (a % 2 == 0) {
    a ++;
  } else {
    a += 3;
  }
} while (a < b);
```

Fig. 6. An example of the application of operation in Fig. 5.

The `abort` operation will ignore any alternative state. It always results in failure and, therefore, can be used (in conjunction with the combiners) to avoid generated some mutants. It is similar to the `false` predicate of Prolog.

The `cut` operation will prune the set of alternative states, in such a way that only the first alternative state will be considered. It is similar to, and was inspired by, the `!` of Prolog.

3.5. $\mathcal{MuDeL}$ combiners

Two or more operations can be combined into a compound operation using the combiners `;;` and `||`. They were inspired in the Prolog operators comma (,) and semi-colon (;).

The first combiner is the *sequence* one, which is represented by `;;` in the $\mathcal{MuDeL}$ syntax. The compound operation `a;;b` incorporates the effects of both `a` and `b`. Every time the operation `a` results in success, the operation `b` is applied. As a side-effects, if the operation `a` does not succeed, the operation `b` will be ignored.

The second combiner is the *alternative* one, which is represented by `||` in the $\mathcal{MuDeL}$ syntax. The compound operation `a||b` indicates that both `a` and `b` are alternative operations for the same purpose. Therefore, the results of either one can appear in a mutant. Actually, the compound operation succeeds every time the operation `a` does and every time the operation `b` does.

The combiners can be used with more than two operations. For instance, we can join three operations as in `a;;b;;c`. Moreover, both combiners can be used together. In this case, the combiner `;;` has a higher precedence than the combiner `||`. The operations can be grouped with double parenthesis to overpass the precedence. For instance, in the compound operation `((a||b));;c`, the operation `c` will be applied to the alternative syntax trees resulting from the operations `a` and `b`.

3.6. $\mathcal{MuDeL}$ modifiers

There are two modifiers that can be applied to an operation. The negation modifier is used to "invert" the result of an operation. It is syntactically represented by a `~` placed in front of the modified operation. Every operation in $\mathcal{MuDeL}$ can result in either a failure or a success. The precise meaning depends on the specific operation on which it is applied. For instance, the `match` operation results in a success if it can unify its operands, and results in failure otherwise. When modified with `~`, a unification will be considered a failure, while the inability to unify will be considered successful.

The *in depth* modifier is used to indicate that the modifier operation should be applied not only to the context tree, but also to every of its subtrees. For instance, when applied to a `match` operation, the unification will be tried with the context tree and with each of its subtrees. Whenever a unification is successful, the `match` operation will result in success and a mutant will be generated. For the `replace` operation, the effects of the modifier is similar. The replacement will be made not only in the context tree, but also in every of its subtrees, in turn. It is important to note that each replacement will take place in the original tree, i.e., after each replacement, the tree is restored to the original one before the next replacement be searched.

3.7. Usage examples

In this section, we illustrate the usage of elements of $\mathcal{MuDeL}$ syntax. We present only the operator body, not including the operators name and the meta-variable declaration sections, once they can be inferred from the operations. Moreover, we give only an informal definition of the semantics of each operation. A formal definition can be found elsewhere [27]. All the examples describe mutant operators for the C language.

An example of a replace operation is presented in Fig. 5, that replaces a **while** statement (matched in Line 1) by a **do–while** one (Line 2). This is the objective of the SWDW mutant operator defined by Agrawal [14].

Assuming that `:p` is bound to the syntax tree of the fragment of C code in Fig. 6(a), after the application the operation in Fig. 5, the code will be replaced by that in Fig. 6(b).

a

```
if (a > 0) {
  while (a < b) {
    a *= 3;
  }
}
```

b

```
while(...) {
  ...
}
while(...) {
  while(...) {
  }
}
```

Fig. 7. An example of **while** statements that will not be replaced by the operation in Fig. 5.

```
:p @@ * replace [S< while( :e ) :s >]
       by      [S< do :s while ( :e ); >]
```

Fig. 8. The replace operation modified by *.

```
int doOpen() {
  if (isClose()) {
    return open();
  } else {
    return 1;
  }
}
```

Fig. 9. A C function example.

```
1 :f @@ match [FD< int :id () :s >]
```

Fig. 10. An example of the matching operation.

```
1   :f @@ * match [FD< while( :e ) :s >]
2 ;;
3   :e @@ * replace [S< :id >]
4            by      [S< 0 >]
```

Fig. 11. A usage example of the combiner ; ;.

Suppose now that :p is bound to the C code in Fig. 7(a). In this case the operation in Fig. 5 will not be applicable, since the **while** statement will form a sub-tree of the whole statement, and, thus, the pattern of the replace operation will not unify to it. Another situation that should be dealt with is illustrated in the C code in Fig. 7. In this case, we have three **while** statements.

To properly deal with this situation, we can modify the replace operation with the modifier *. The meaning of a replace operation with the modifier * is that every successful matching of *b* with *c* itself or any of its subtrees will produce a mutated tree. Indeed, we can think of this modified operation as producing alternative states, and each of such states will have its own execution flow and eventually producing a mutant. Therefore, a more adequate mutant for the SWDW is the one presented in Fig. 8.

Suppose that the meta-variable :f is bound to the code in Fig. 9. Then, after the application of the matching operation in Fig. 10, the meta-variable :s will be bound to the body of the function. Observe that the pattern will match a function with no arguments that returns an int value.

The compound operation in Fig. 11 will replace every variable in the control expression of a **while** statement to 0. It is important to note that, for every such a control expression, the matching will produce an alternative state and the replace operation in Line 3 will be applied to each one, possibly generating more alternative states by itself. This example illustrates the usage of the match operation to constrain the context in which the replacement should be applied. In this case, the operator was designed to be applicable only to **while** statement control expressions. Suppose now that one wants the replacement to be applied to control expression of every iterate statement, i.e., every **while**, **do–while** or **for** statement. In other words, we want to join the set of

```
1  ((
2    :f @@ * match [FD< while( :e ) :s >]
3  ||
4    :f @@ * match [FD< do :s while( :e ) ; >]
5  ||
6    :f @@ * match [FD< for ( :init: ; :e :inc ) :s >]
7  ))
8  ;;
9    :e @@ * replace [S< :id >]
10            by      [S< 0 >]
```

Fig. 12. A usage example of the combiner ||.

```
:f @@ * ((
  match [FD< while( :e ) :s >]
||
  match [FD< do :s while( :e ) ; >]
||
  match [FD< for ( :init: ; :e :inc ) :s >]
))
;;
  :e @@ * replace [S< :id >]
           by      [S< 0 >]
```

Fig. 13. An example of the usage of parenthesis to factor out common modifiers.

```
  :s @@ * replace [E< :id >]
           by      [E< 0 >]
;;
  :id @@ consult type with :id [T< int >]
```

Fig. 14. Example of `consult` operation.

```
  * match [ID< :id1 >]
;;
  * replace [ID< :id2 >]
    by      [ID< :id1 >]
;;
  :id1 @@ ~ match [ID< :id2 >]
```

Fig. 15. Example of operations that exchanges an identifier by every other identifier.

alternative states of three matching operations. This can be done with the alternative combiner, which is represented by || in the $\mathscr{MuDeL}$ syntax. For instance, the compound operation in Fig. 12 will achieve the objective. (The parenthesis are necessary because ;; has a higher precedence than ||.)

The parenthesis can also be used to avoid explicitly declaring the context tree in every operation, as well as the * modifier. Therefore, the compound operation in Fig. 12 is equivalent to the compound operation in Fig. 13. There is, however, a small difference between both w.r.t. the efficiency and the order in which the alternative states (and, hence, the eventual mutants) are produced. While in Fig. 12 :f is traversed three times, since for each match operation starts from the root node of the :f syntax, in Fig. 13 :f is traversed only once.

The other two basic operations (namely, **assert** and **consult**) are related to context-sensitive mutants. The `consult` operation in Fig. 14 is used to ensure that only identifiers with the `int` attribute is mutated to `0`.

To illustrate the use of the assert operation, consider a mutant operator that exchanges each identifier by each distinct identifier in the artifact. (For sake of simplicity, we assume that this mutation can be carried out without taking context into consideration.) Firstly, consider the operator in Fig. 15, which will exchange each identifier matched in Line 3 by the identifier

```
  * match [ID< :id1 >]
;;
  ~ consult used with :id1
;;
  assert used with :id1
;;
  * replace [ID< :id2 >]
    by      [ID< :id1 >]
;;
  :id1 @@ ~ match [ID< :id2 >]
```

Fig. 16. Example of usage of `consult` and `assert` operations to avoid employing the same identifier more than once.

```
:f @@ * ((
  match [FD< while( :e ) :s >]
||
  match [FD< do :s while( :e ) ; >]
||
  match [FD< for ( :init: ; :e :inc ) :s >]
))
;;
  :e @@ * replace [S< :id >]
           by      [S< 0 >]
;;
  cut
```

Fig. 17. Usage of `cut` operation.

matched in Line 1 that are not equal to each other (ensured by the negated matching in Line 6). If the same identifier occurs in more than one location, the match in Line 1 will produce an alternative state for each one, and the replacement will generate several identical mutants. We canavoid this situation with the usage of `consult` and `assert` operations, as illustrated in Fig. 16. In this way, the `consult` operation in Line 3 ensures that there is no tuple for the `used` attribute with the `:id1` value. Then, if so, the assert operation stores such a tuple in the root node (since no context pattern was furnished).

The `cut` operation can be used to prune the set of alternative states that the previous operations might have generated. It was introduced in $\mathscr{MuDeL}$ language for sake of completeness, since the other operations are inspired in Prolog, and this language has the cut operator (!), whose purpose is similar. An example of the usage of cut is presented in Fig. 17, which equals the example in Fig. 13, expect for the `cut` operation added in the end. The effect is that after the `cut` operation is executed, any pending alternative states are forgotten, i.e., at most one mutant will be generated.

In Fig. 18 we present the SDWE operator that is meant to change every **while** statement into a **do–while** and also change the control expression into 0 and 1. This kind of mutant is usually necessary when branch coverage is required. Observe that, in a C program, changing the control expression into 0 has the effect of iterating the body of the **while** statement exactly once, whereas changing the control expression into 1 converts the **do–while** into an infinite loop. Fig. 19(a) presents a simple C program and Figs. 19(b)–(e) present the mutants that will be generated for this program with the SDWE operator.

The replacing operation in Lines 5 and 6 changes every **while** statement into a **do–while** statement, in any depth. The meta-variable `:e` stands for the control expression of the **while**. The group of operations in Lines 8–20 makes changes in this control expression. Observe that the context pattern declaration in Line 8 affects the whole group, and, consequently, every operation therein.

The (negated) matching in Line 9 makes sure that the context pattern (`:e`, in this case) is not equal to 0. If so, the context pattern is changed to 0, by the replacement in Lines 11 and 12, and a mutant is generated. Note that these two operations compose a sequence, which is part of an alternative list. Then, the next alternative is tried, in this turn w.r.t. the expression 1. Finally, the operation in Line 19 is tried and a mutant is generated only with the replacement of Line 5.

Analyzing how the mutants are generated in this example illustrates the way $\mathscr{MuDeL}$ processes a mutant operator definition. The replacing operation (Lines 5 and 6) is marked with the *in depth* modifier and, therefore, the whole program syntax tree will be scanned, looking for nodes that match the respectivepattern and changing them accordingly. The replacing operation and the group of operations in Lines 8–20 compose a sequence, i.e., every mutant should include the effects of the replacing *and* the effects (if any) of the group. This group, by its turn, is composed by a list of three alternatives: the first alternative is in Lines 9–12; the second one is in Lines 14–17; and the last one is in Line 19. Only the effects (if any) of one of these alternatives will be included

```
operator SDWE
  var :e [expression]
      :s [statement]
begin
  * replace [statement< while ( :e ) :s >]
    by      [statement< do :s while ( :e ) ; >]
;;
  :e @@ ((
       ~ match [expression< 0 >]
    ;;
      replace [expression]
      by      [expression< 0 >]
  ||
       ~ match [expression< 1 >]
    ;;
      replace [expression]
      by      [expression< 1 >]
  ||
    donothing
  ))
end operator
```

Fig. 18. A multi-purpose **while** mutant operator.

in a particular mutant. For instance, Mutants #1 and #2 in Figs. 19(b)–(c) are generated by replacing the outermost **while** of the program in Fig. 19(a) and applying the first and the third alternatives, respectively. (Observe that the second alternative does not generate a mutant, since the operation in line 14 does not succeed.) On the other hand, Mutants #3–#5 in Figs. 19(d)–(f) are generated by replacing the innermost while and applying each of the alternatives, respectively.

4. Applying $\mathscr{M}u\mathscr{D}e\mathscr{L}$

The usefulness of $\mathscr{M}u\mathscr{D}e\mathscr{L}$ can be measured by its suitability in defining mutants, which is its primary goal; in allowing the reuse of mutant operators in different languages; and in generating a mutant generator prototype module that can be evolved and incorporated into a mutation-testing environment. Currently, we have already described mutant operators for (i) specifications written in colored Petri nets (CPNs) and FSMs, (ii) for the functional language standard meta-language (SML), and (iii) for traditional languages such as C, C++ and Java.

In this section, we present our experience using $\mathscr{M}u\mathscr{D}e\mathscr{L}$, and bring some evidence of its usefulness. However, more studies are necessary in order to soundly validate the language. Moreover, there is a lack of feedback from other research groups, and a thorough validation would involve the use of the language by others.

FSM, CPN and SML mutant operators: We used $\mathscr{M}u\mathscr{D}e\mathscr{L}$ to define mutants for specifications written in FSMs [36] and CPNs [37]. In the case of FSMs, we use $\mathscr{M}u\mathscr{D}e\mathscr{L}$ to describe the nine mutant operators defined by Fabbri et al. [38]. Using the `mudelgen` system (described in the next section), we were able to use the mutant operators within the Plavis/FSM environment, which are being used in experiments in Brazilian National Space Agency, in the scope of a project supported by CNPq and CAPES-Coffecub.[2] In the case of CPNs, we observed that the language was useful to allow a rapid prototyping and experimentation with different kinds of mutants. In the whole, 29 mutant operators were defined and used in Proteum/CPN tool [37]. It is important to mention that CPNs annotation language is based on SML and we could reuse some common parts of the mutant operator descriptions in both languages [39].

C and C++ mutant operators: For C language, we described the 77 mutant operators proposed by Agrawal [14], which are implemented in Proteum/C tool. Next, we adapted these operators and described similar mutant operators for C++. We realized that, by carefully designed the grammar and the $\mathscr{M}u\mathscr{D}e\mathscr{L}$ definition, we could reuse 65 operators, nearly 85% of them. To illustrate how this was possible, consider again the operator in Fig. 5. Examining the definition, we can observe that the only relation between the operator and the language of the artifact is in the patterns. Even in the patterns, only the types of the patterns and of the meta-variables and the sequence of terminal symbols are relevant. Therefore, the same definition can be applied both for C and C++, provided that the respective grammars agree in these points: (i) the name of the relevant non-terminal symbols

[2] http://www.labes.icmc.usp.br/plavis/.

a

```
int main() {
  while( 1 ) {
    int c;
    if(( c = getchar() ) == EOF)
      break;
    while ( c > 'A' ) c--;
  }
}
```

b

```
int main() {
   do {
    int c;
    if(( c = getchar() ) == EOF)
      break;
    while( c > 'A' ) c --;
   } while( 0 );
}
```

c

```
int main() {
   do {
    int c;
    if(( c = getchar() ) == EOF)
      break;
    while ( c > 'A' ) c --;
   } while( 1 );
}
```

d

```
int main() {
  while( 1 ) {
    int c;
    if(( c = getchar() ) == EOF)
      break;
     do  c --;  while( 0 );
  }
}
```

e

```
int main() {
  while( 1 ) {
    int c;
    if(( c = getchar() ) == EOF)
      break;
     do  c --;  while( 1 );
  }
}
```

f

```
int main() {
  while( 1 ) {
    int c;
    if(( c = getchar() ) == EOF)
      break;
     do  c --; while( c > 'A');
  }
}
```

Fig. 19. (a) Original program. (b)–(e) Mutants generated by operator in Fig. 18. The mutated parts of the code are highlighted.

(that define the types available) and (ii) the sequence of terminal symbols that appear in the relevant non-terminal productions. Observe that not the whole sequence of terminal symbols should be the same. Rather, only the terminal symbols that are relevant to the $\mathscr{MuDeL}$ definition. In the SDWD example, for the mutant definition to be applicable, it is necessary that both grammars have a non-terminal symbol S and that there is a derivation from S to '`while`' '`(`' ⟨*E*⟩ '`)`' ⟨*S*⟩ and to '`do`' ⟨*S*⟩ '`while`' '`(`' ⟨*E*⟩ '`)`' '`;`'.

Java mutants: We have also carried out a study, in which we try to apply the mutant descriptions for C and C++ to the Java language. Since the grammar of these languages are similar, we could observe that 31 mutant operators could be reused for Java. The results of the application to C, C++ and Java are summarized in Fig. 20.

We have investigated how $\mathscr{MuDeL}$ can be used with mutant operators that are more semantic-driven. We have described the class mutant proposed by Kim et al. [40,41]. Some of those operators are related to the semantics of inheritance, overloading and overriding concepts, which varies from one OO language to another. Those complex operators can only be described with complex $\mathscr{MuDeL}$ code. Indeed, the complexity is inherent to the operators and, to our knowledge, their definition could only made simpler if we hide the complexity in a more complex operation of the language. The same situation occurs with any language. There is a trade-off between the simplicity of the operations and its ability to handle complex mutants. The complexity of these kinds of mutants comes from the underlying semantics of those languages. For instance, let us consider CM operators defined by Kim et al. [41]. From the 20 mutant operators, we could describe easily 10 of them, since they require only syntax driven changes. Other five could be described, provided that some semantic information is collected and is available to consult operations. However, this semantic should be coded, anyway, and to adequately capture the semantics of OO languages is not an easy task and is still open issue in the programming language semantics research. Unfortunately, we could not find an easy way to tackle this problem, either. The remaining five ones might be described, depending on what exactly the authors mean. For instance, FAR operator is defined as "Replace a field access expression with other field expressions of the same field name". Unfortunately, it is not clear what "a field access expression" exactly means. Other example of ambiguity is the definition of AOC operator. In Kim et al. [41], the authors defined it as "Change a method argument order in method invocation expressions".

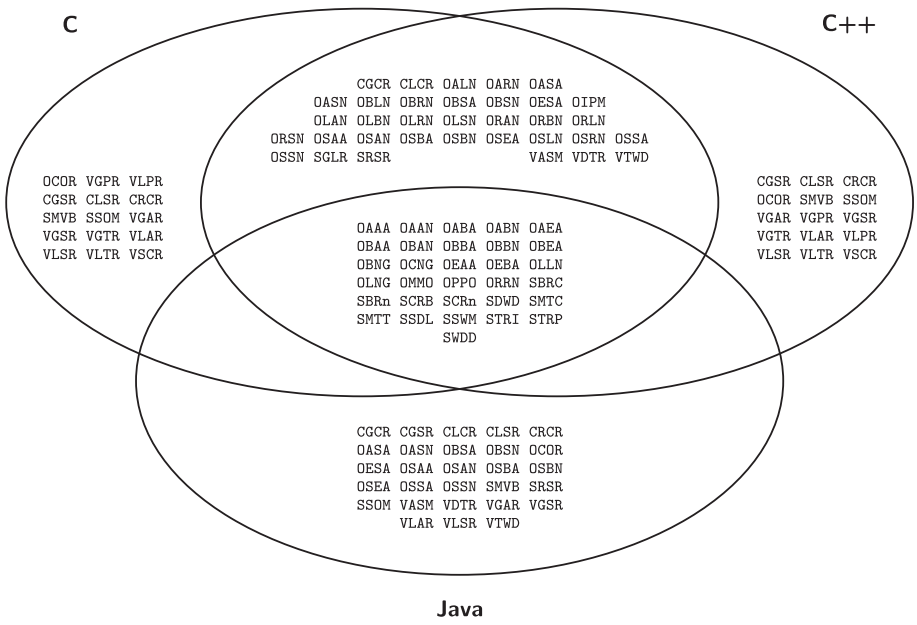


Fig. 20. Mutant definition reuse for C, C + + and Java.

It is not clear how many mutants can be generated from a method with more than two arguments. The possibilities are: (i) one mutant for each permutation of the arguments (i.e., exponentially many mutants); (ii) one mutant for every shift, in the same vein as Agrawal's SSOM mutation operator. In Kim et al. [40], the authors give a little bit more explanation about the operator and provide an example, neither of which clarify the point. These ambiguities evidence the necessity of a formal definition of mutant operator.

5. mudelgen

In order to be able to process the $\mathscr{MuDeL}$ descriptions, we implemented the `mudelgen` system. In this section we discuss its main implementation aspects. Suppose that we are interested in describing mutant operators for a language *L*. The first step is to obtain a grammar *G* for *L*. When `mudelgen` is input with *G*, it produces a program `mudel.G`. This program can then be run with a mutant operator definition *OD* and an artifact *P*. After checking whether both *OD* and *P* are syntactically correct w.r.t. the input grammar *G*, a mutant set *M* is generated.

To manipulate the `mudelgen` input grammar, we use `bison` and `flex`, which are open source programs similar to, respectively, `yacc` and `lex` [42]. Although these tools ease the task of manipulating grammars, they, on the other hand, restrict the set of grammars that `mudelgen` can currently deal with to LALR(1) grammars [34,35,42]. The grammar input to `mudelgen` is provided in two files: the `.y` and the `.l`. The `.y` file is the context-free grammar, written in a subset of `yacc` syntax [42]. The `.l` file is a lexical analyzer and gives the actual form of the terminal symbols of the grammar and it is encoded in a subset of the `lex` syntax [42]. The attribute values are attached to the tree nodes with special C functions put in the semantic action of the productions of `.y`. For instance, the function `assertFact` can be used to store an attribute value in a way similar to the `assert` operation.

The `mudelgen` is divided into two parts: one part with the elements that depend on the input grammar and the other one with elements that do not. Fig. 21 depicts how these parts interact and illustrates the overall execution schema of `mudelgen`. The grammar-dependent part is actually composed by three modules, which are executable programs: `treegen`, `opdescgen` and `linker`. The grammar-independent part is embodied in the `Object Library`. The major portion of the `Object Library` is devoted to the so-called MuDeL Kernel, which is responsible for interpreting the mutant operator definition and manipulating the syntax tree accordingly. The remaining units in the `Object Library` allow the communication between the MuDeL Kernel and the external modules $\mathscr{MuDeL}$ Animator and DS Oracle, described later.

The units that depend on the grammar are built either by `treegen` or by `opdescgen`. Module `treegen` analyzes *G* and generates the units: (i) `STP` (syntax tree processor), which is responsible to syntactically analyze a source product *P* into a syntax tree and (ii) `Unparse`, which is responsible to convert the mutated syntax trees into the actual mutants. Module `opdescgen` analyzes *G* and generates the unit `ODP` (operator description processor), which analyzes a mutant operator description *OD* w.r.t. *G* and generates an abstract representation of how to manipulate the syntax tree in order to produce the mutants. Finally, the `linker` module will link all these grammar-depending units and the appropriate portion of the `Object Library` and generate the program `mudel.G`.

The program `mudel.G` is input with a source product *P* and a mutant operator definition *OD*. These input data are processed by `STP` and `ODP`, respectively, and handled by MuDeL Kernel. During its execution, MuDeL Kernel will generate one or more

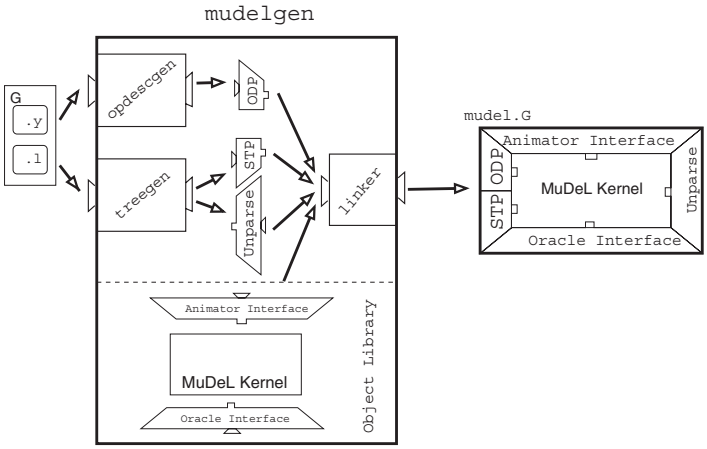


Fig. 21. mudelgen execution schema.

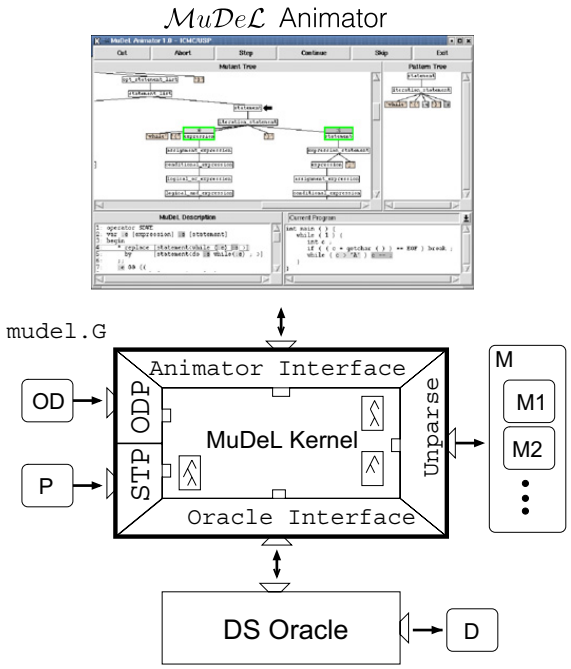


Fig. 22. Execution of `mudel.G`.

mutated syntax trees, which are processed by `Unparse` in order to generate the actual mutants. `Unparse` can output the generated mutants in several formats. Currently, the mutants can be (i) sent to standard output; (ii) restored in SQL databases (e.g., MySQL); or (iii) written to ordinary files (each mutant in a separate file). Optionally, the DS Oracle can be used to check whether the mutants were correctly generated (see Section 5.1). The execution of the program `mudel.G` can be visually inspected with the $\mathcal{MuDeL}$ Animator (see Section 5.2). The overall execution schema of `mudel.G` is depicted in Fig. 22.

5.1. Denotational semantics-based oracle

The number of mutants generated is often very large and manually checking them is very costly and error-prone. Therefore, the validation of `mudelgen` is a hard task, mainly due to the amount of output which is produced. To cope with this problem, we adopted an approach that can be summarized in two steps. Firstly, we employed *denotational semantics* [25] to formally define the semantics of $\mathcal{MuDeL}$ language [27]. Secondly, supported by the fact that denotational semantics is primarily based on lambda calculus, we used the language SML [43], which is also based on lambda calculus, to code and run the denotational semantics of $\mathcal{MuDeL}$. We implemented an external module DS Oracle that can be run in parallel with `mudel.G` through the `Oracle Interface` in a *validation* mode. When invoked, the DS Oracle receives the information about a mutant operator *OD*

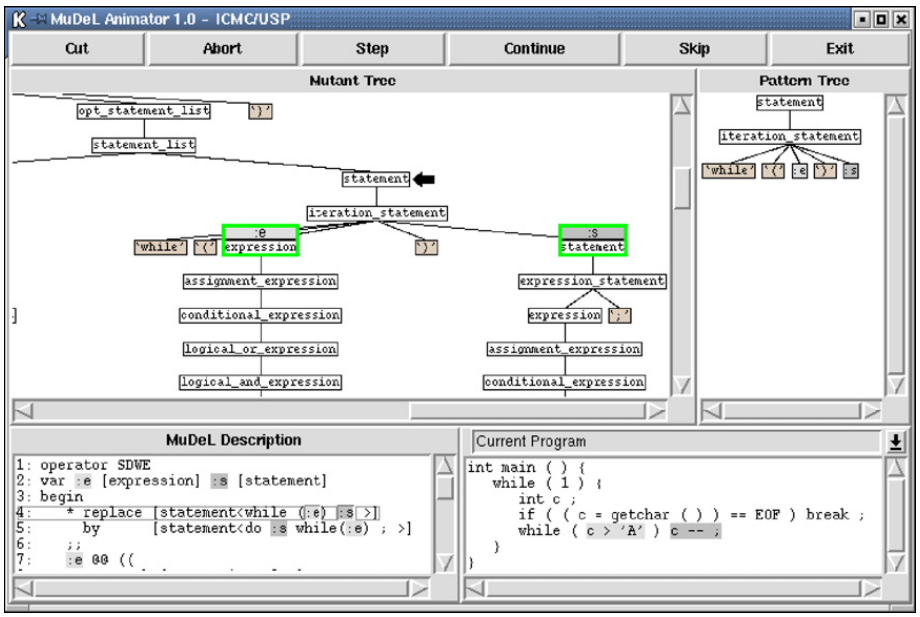


Fig. 23. $\mathcal{MuDeL}$ Animator main window.

and derives a denotational function μ (in the mathematical sense) that formally defines the semantics of *OD*. Then, the DS Oracle reads the information about the source product *P* and the set of generated mutants *M*. The mutants in *M* are compared with the mutants defined by μ. Any identified difference is reported in the discrepancy report *D*.

It is important to remark that the *validation* mode has no usefulness for users interested in `mudelgen`'s functionalities, since it brings no apparent benefit. However, it is very useful for validation purpose, since it improves the confidence that the mutants are generated in the right way. Nonetheless, from a theoretical viewpoint, there is a possibility that a fault in the implementation be not discovered, due to the fact that the SML implementation may also possess a fault that makes it produces the same incorrect outputs. However, the probability that this occurs in practice is very small. Both languages (i.e., C++ and SML) are very different from one another. Moreover, the algorithms and overall architectures of both implementations are very distinct. While we employed an imperative stack-based approach in C++, we extensively used continuation and mappings [25] in SML. Consequently, it is not trivial to induce the same kind of misbehavior in both implementations. In other words, although none of them is fault free, the kind of faults they are likely to include is very distinct. With this consideration, we conclude that the use of denotational semantics and SML was a powerful validation mechanism for `mudelgen`.

5.2. $\mathcal{MuDeL}$ animator

We have also implemented a prototyping graphical interface—called $\mathcal{MuDeL}$ Animator—for easing the visualization of the execution of a mutant operator. $\mathcal{MuDeL}$ Animator was implemented in Perl/Tk and currently has some limited features that allows inspecting the log of execution, without, however, being able to interfere in the process. Since $\mathcal{MuDeL}$ Animator enables us to observe the execution of a mutant operator definition, it is very useful not only for obtaining a better understanding of the $\mathcal{MuDeL}$'s mechanisms, but also for (passively) debugging a mutant operator.

Fig. 23 presents the main window of $\mathcal{MuDeL}$ Animator, where the example of Fig. 18 was loaded and is being executed. At the top of the window are the buttons that control the execution of the animator, such as *Step*, *Exit*, etc. The remaining of the window is divided up into four areas:

$\mathcal{MuDeL}$ description: In the left bottom area, $\mathcal{MuDeL}$ Animator presents the mutant operator definition. A rectangle indicates which line is currently executing. Every meta-variable is highlighted with a specific color. The same color is used in whichever occurrence of the same meta-variable throughout all the other areas.

Mutant tree: In the left top area, the animator shows the syntax tree of the product, reflecting any change so far accomplished by the execution. An arrow indicates which node is currently the context tree. Meta-variable bindings are presented by including the names of the meta-variable above the respective tree nodes.

Current product: In the right bottom area, the current state of the product, obtained by traversing the current state of the mutant tree, is presented. The parts in the mutant that correspond to the nodes bound to meta-variables are highlighted with the respective color.

Pattern tree: In the right top area, $\mathcal{MuDeL}$ Animator shows the tree of the pattern currently active (i.e., in the current line) in the $\mathcal{MuDeL}$ description area.

Since $\mathcal{MuDeL}$ Animator enables us to observe the execution of a mutant operator description, it is very useful not only for obtaining a better understanding of the $\mathcal{MuDeL}$'s mechanisms, but also for (passively) debugging a mutant operator.

5.3. Operational aspects

We have designed `mudelgen` to generate a module for each particular language. These modules might be used as a standalone tool, or as an component within a larger, more complete upper-level environment. Therefore, we delegate some common tasks of managing mutants to this upper-level environment, such as (i) preparing the source code; (ii) selecting parts of the source code to which the operators should be applied; (iii) selecting which of the generated mutants should be used or discarded, and so on. We decided to keep these tasks to the upper-level as a way of increasing flexibility. In this way, the module could be used in different context and with different purpose, such as constraint mutation [44], essential mutation [11] and so on. However, this upper-level environment is very important to make the application of mutant testing feasible. For instance, the module `mudelgen` will apply the mutant operator to the whole source code that is provided as input. For large source codes, this could be impractical or even impossible.

Besides being generated, the mutants must be executed in some way, in order to collect the results and decide whether a mutant was killed by a test set or would stay alive. In general, the execution of mutants can be very costly. In the particular case of mutants of programs, it may be necessary to compile and execute every mutant. To tackle this problem, some researchers have proposed alternative schemas of execution. For instance, Untch et al. [45] use mutant schemata. Several mutants are put in a single generic source, which is parameterized to behave like any of the individual mutants. In this way, only one compilation for each schemata is necessary. A similar approach is employed by Delamaro et al. [18] in Proteum/IM to avoid compiling every mutant in a separate file.

Considering $\mathcal{MuDeL}$ as a language to define mutants, any of these approaches can be used, although the most natural one is the individual compilation schema. For instance, a specialized $\mathcal{MuDeL}$ compiler can be constructed, which will generate one or more mutant schematas instead of individual mutants. However, it is necessary to take into account the semantics of the target language, since the way several mutants can vary from one language to another. This is an interesting point for future research. Nonetheless, it is important to highlight that a language like $\mathcal{MuDeL}$ can help in this context, providing a uniform notation and precise semantics for describing the construction of each mutant.

6. Concluding remarks

The efficacy of mutation testing is heavily related to the quality of the mutants employed. Mutant operators, therefore, play a fundamental role in this scenario, because they are used to generate the mutants. Due to their importance, mutant operators should be precisely defined. Moreover, they should be experimented with and improved. However, implementing tools to support experimentation and validation of the mutant operators before delivering a mutant environment is very costly and time-consuming.

In this paper we presented the $\mathcal{MuDeL}$ language as a device for describing mutant operators and generating a mutant generator prototyping module. The language is based on the transformational paradigm and also uses some concepts from logical programming. Being defined in $\mathcal{MuDeL}$, an operator can be "compiled" and the respective mutants can be generated using the `mudelgen` system. $\mathcal{MuDeL}$ and `mudelgen` together form a powerful mechanism to develop mutant operators. The mutant operators can be validated either formally (with the facilities of DS Oracle) or manually (with the facilities of $\mathcal{MuDeL}$ Animator).

The design decisions we have taken lead us one step further towards the achievement of our goals. There are some points that need to be further investigated. For instance, $\mathcal{MuDeL}$ was mainly designed to deal with context-free mutations. With this decision, we keep the language quite simple, yet considerably expressive. However, there are some important kinds of mutants that are inherently context-sensitive. For example, some program mutant operators might need knowledge that are not readily available, such as the set of variables defined prior to a specific point in the program, or whether a method overrides the method of a parent class. Although, these situations can be dealt with `assert` and `consult` operations, we realize that these mutant operators are not easy to write nor are the definition easy to follow. Indeed, the problem of dealing with context aware transformation is a hard problem for any transformational language [33]. We are still investigating how these situations can be more suitably handled. For instance, we observe some idioms in the mutant operators that might be candidates to be included in the language as primary operations.

The experiments we have carried out with $\mathcal{MuDeL}$ involved languages for which there were supporting tools, namely Petri nets and C programs. Although fully useful in demonstrating its potential usage, these experiments are not a complete validation. Right now, we are working on a project where Java mutant operators are being described, what will further contribute towards the validation of the ideas presented herein.

There are tasks that are hard, cumbersome or even impossible to be carried out only with the construction $\mathcal{MuDeL}$ embodies. As example, we can cite arithmetics and string manipulation. To tackle this problem, we are currently developing an API (application programming interface) to allow the implementation and inclusion of *built-in* rules written in a conventional programming language, namely, in C++. We, then, keep the kernel of $\mathcal{MuDeL}$ tiny, whereas built-in rules can be provided for any further need we have to take care of.

Some forthcoming steps in this research include:

- To develop an integrated development environment (IDE), providing features to edit the context-free grammar, the mutant operator and the original product in a manner as uniform as possible, and also providing features to compile, execute and debug the mutant operator definition `mudelgen` is currently operated by means of command-line invocations and has some limited graphical interaction (with $\mathcal{MuDeL}$ Animator). To ease the usage and experimentation, an IDE would be more appropriate.
- To further investigate the context-sensitiveness of some kinds of mutants and devise constructions to cope with them.
- To integrate the $\mathcal{MuDeL}$ and the `mudelgen` in a complete mutation tool. Mutation testing demands also other activities such as test case handling, mutation execution, result analysis, and so on. We are now specifying and designing a complete mutation tool which follows the main ideas of $\mathcal{MuDeL}$, i.e., a tool with multi-language support.
- To investigate the relationship between syntax and semantic mutation.

References

[1] Budd AT. Mutation analysis: ideas, examples, problems and prospects. Computer program testing. Amsterdam: North-Holland Publishing Company; 1981. p. 129–148.
[2] DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: help for the practicing programmer. IEEE Computer 1978;11(4):34–41.
[3] Fabbri SCPF, Maldonado JC, Sugeta T, Masiero PC. Mutation testing applied to validate specifications based on statecharts. In: ISSRE—international symposium on software reliability systems. 1999. p. 210–9.
[4] Fabbri SCPF, Maldonado JC, Masiero PC, Delamaro ME. Mutation analysis for the validation of Petri net based specifications. In: 8th workshop of software quality, Curitiba, PR, 1994 [in Portuguese].
[5] Souza SRS, Maldonado JC, Fabbri SCPF, Lopes de Souza W. Mutation testing applied to Estelle specifications. Quality Software Journal 2000;8(4):285–301 [publicado também no 33rd Hawaii Internacional Conference on System Sciences, 2000].
[6] Sugeta T, Maldonado JC, Wong WE. Mutation testing applied to validate SDL specifications. In: Hierons RM, Groz R, editors, 16th IFIP international conference on testing of communicating systems—TestCom2004, Oxford, UK, 2004.
[7] Kovács G, Pap Z, Viet DL, Wu-Hen-Chang A, Csopaki G. Applying mutation analysis to SDL specifications. In: Reed R, Reed J, editors, SDL 2003: system design—11th SDL forum. Lecture notes on computer science, vol. 2708, Springer, Heidelberg, Stuttgart, Germany, 2003. p. 269–84.
[8] Probert RL, Guo F. Mutation testing of protocols: principles and preliminary experimental results. In: Proceedings of the IFIP TC6 third international workshop on protocol test systems. Amsterdam: North-Holland, 1991. p. 57–76.
[9] Ritchey RW. Mutating network models to generate network security test cases. In: Mutation 2000, San Jose, California, 2000. p. 101–8.
[10] Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? In: International conference on software engineering. St. Louis, MO, USA, 2005. p. 402–11.
[11] Offutt AJ, Rothermel G, Zapf C. An experimental evaluation of selective mutation. In: 15th international conference on software engineering. Baltimore, MD: IEEE Computer Society Press; 1993. p. 100–7.
[12] Wah KSHT. A theoretical study of fault coupling. Software Testing, Verification and Reliability 2000;10(1):3–45.
[13] Nakagawa EY, Maldonado JC. Software-fault injection based on mutant operators. In: Anais do XI Simposio Brasileiro de Tolerancia a Falhas, Florianopolis, SC, 2001. p. 85–98.
[14] Agrawal H. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center/Purdue University; March 1989.
[15] DeMillo RA, Guindi DS, King KN, McCracken WM, Offutt AJ. An extended overview of the Mothra testing environment. In: Second workshop on software testing, verification and analysis, Baniff, Canadá, 1988.
[16] Kim S, Clark JA, Mcdermid JA. The rigorous generation of Java mutation operators using HAZOP. In: 12th international conference on software & systems engineering and their applications (ICSSEA'99), 1999.
[17] Ma Y-S, Kwon Y-R, Offutt J. Inter-class mutation operators for Java. In: 13th international symposium on software reliability engineering—ISSRE'2002, Annapolis, MD, 2002.
[18] Delamaro ME, Maldonado JC, Mathur AP. Interface mutation: an approach for integration testing. IEEE Transactions on Software Engineering 2001;27(3): 228–47.
[19] Alexander RT, Bieman JM, Ghosh S, Ji B. Mutation of Java objects. In: 13th International symposium on software reliability engineering, 2002. p. 341–51.
[20] Fabbri SCPF, Maldonado JC, Delamaro ME, Masiero PC. Proteum/FSM: a tool to support finite state machine validation based on mutation testing. In: XIX SCCC—international conference of the Chilean computer science society, Talca, Chile, 1999. p. 96–104.
[21] Vincenzi AMR, Nakagawa EY, Maldonado JC, Delamaro ME, Romero RAF. Bayesian-learning based guidelines to determine equivalent mutants. International Journal of Software Engineering and Knowledge Engineering 2002;12(6):675–90.
[22] Vincenzi AMR, Simão AS, Delamaro ME, Maldonado JC. Muta-pro: towards the definition of a mutation testing process. Journal of Brazilian Computer Society 2006;12(2):47–61.
[23] Neighbors J. The Draco approach to constructing software from reusable components. IEEE Transactions on Software Engineering 1984;10(5):564–74.
[24] Bratko I. Prolog programming for artificial intelligence. 2nd ed., Wokingham, England, Reading, MA: Addison-Wesley; 1990.
[25] Allison L. A practical introduction to denotational semantics. Cambridge, UK: Cambridge University Press; 1986.
[26] Stoy JE. Denotational semantics: the Scott–Strachey approach to programming language theory. Cambridge, MA: MIT Press; 1977.
[27] Simão AS, Maldonado JC, Bigonha RS. Using denotational semantics in the validation of the compiler for a mutation-oriented language. In: Proceedings of 5th workshop of formal methods, Gramado, RS, 2002. p. 4–19.
[28] Weyuker EJ. On testing non-testable programs. Computer Journal 1982;25(4):465–70.
[29] Delamaro ME, Maldonado JC, Mathur AP. Proteum: a tool for the assessment of test adequacy for C programs: user's guide. Technical Report SERC-TR-168-P, Software Engineering Research Center, Purdue University, West Lafayette, IN; April 1996.
[30] Fabbri SCPF. Mutation analysis in the context of reactive systems: a contribution to establishing testing and validation strategies. PhD thesis, IFSC/USP, São Carlos, SP; 1996 [in Portuguese].
[31] McDermid JA, Pumfrey DJ. A development of hazard analysis to aid software design. In: Compass'94: 9th annual conference on computer assurance. Gaithersburg, MD: National Institute of Standards and Technology; 1994. p. 17–26.
[32] Cordy JR, Carmichael IH, Halliday R. The TXL programming language—version 8. Technical Report, Department of Computing and Information Science, Queen's University, Kingston, Canada; April 1995.
[33] Cordy JR, Dean T, Malton A, Schneider K. Source transformation in software engineering using the txl transformation system. Technical Report 13; 2002.
[34] Salomaa A. Formal languages. New York: Academic Press; 1973.
[35] Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques and tools. Reading, MA: Addison-Wesley; 1985.
[36] Simão AS, Ambrosio AM, Fabbri SCPF, Amaral AS, Martins E, Maldonado JC. Plavis/FSM: an environment to integrate FSM-based testing tools. In: Caderno de Ferramentas do XIX Simposio Brasileiro de Engenharia de Software, Uberlandia, MG, 2005.

[37] Simão AS. Mutation analysis application in the context of testing and validation of coloured petri nets. PhD thesis, ICMC/USP, São Carlos, SP; 2004.
[38] Fabbri SCPF. A análise de mutantes no contexto de sistemas reativos: Uma contribuição para o estabelecimento de estratégias de teste e validação. PhD thesis, IFSC/USP, São Carlos, SP; 1996.
[39] Yano T, Simão AS, Maldonado JC. Estudo do teste de mutação para a linguagem standard ML. In: Solar M, Fernandez-Baca D, Cuadros-Vargas E, editors, 30ma Conferencia Latinoamericana de Informatica (CLEI2004). Sociedad Peruana de Computacion, 2004. p. 734–44, iSBN 9972-9876-2-0.
[40] Kim S, Clark JA, Mcdermid JA. Class mutation: mutation testing for object-oriented programs. In: Object-oriented software systems, Net.ObjectDays'2000, Erfurt, Germany, 2000.
[41] Kim S, Clark JA, Mcdermid JA. Object-oriented testing strategies using the mutation testing. Software Testing Verification and Reliability 2001;11:207–25.
[42] Mason T, Brown D. Lex & Yacc. O'Reilly; 1990.
[43] Hansen MR, Rischel H. Introduction to programming using SML. Reading, MA: Addison-Wesley; 1999.
[44] Wong E, Maldonado JC, Delamaro ME, Mathur AP. Constrained mutation for C programs. In: 8° Simposio Brasileiro de Engenharia de Software, Curitiba, Brasil, 1994. p. 439–52.
[45] Untch R, Harrold MJ, Offutt J. Mutation analysis using mutant schemata. In: International symposium on software testing and analysis, Cambridge, MA, 1993. p. 139–48.

Apêndice H

A. L. Bonifacio, A. Moura, A. S. Simão, J. C. Maldonado. Towards Deriving Test Sequences by Model Checking. Electronic Notes in Theoretical Computer Science, v. 195, p. 21-40, 2008.

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 195 (2008) 21–40
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Towards Deriving Test Sequences by Model Checking

Adilson Luiz Bonifácio[1] Arnaldo Vieira Moura[2]

Computing Institute, University of Campinas
P.O. 6176 – Campinas – Brazil – 13081-970

Adenilso da Silva Simão[3] José Carlos Maldonado[4]

Mathematic Science and Computing Institute, University of São Paulo
P.O. 668 – São Carlos – Brazil – 13560-970

Abstract

Model-based testing automatically generates test cases from a model describing the behavior of the system under test. Although there exist several model-based formal testing methods, they usually do not address time constraints, mainly due to the fact that some supporting formalisms do not allow a suitable representation of time. In this paper, we consider such constraints in a framework of Timed Extended Finite State Machines (TEFSMs), which augment the Extended Finite State Machine (EFSM) model by including a notion of explicit and implicit time advancement. We use this extension to address conformance testing by reducing the confirming configuration problem to the problem of finding a path in a TEFSM product.

Keywords: Model Checking, Timed EFSM, Conformance testing, Suspicious Configuration.

1 Introduction

Model-based testing comprises the automatic generation of efficient test cases using models of system requirements, usually based on formally specified system functionalities. It involves the (i) construction of a suitable formal model, (ii) derivation of test inputs, (iii) calculation of test outputs, (iv) execution of test inputs over implementations, (v) comparison of the results from the calculated test outputs and the implementation executions, and (vi) decision of whether the testing should be stopped. All these tasks are tightly related to each other. For instance, the

[1] Email: adilson@ic.unicamp.br
[2] Email: arnaldo@ic.unicamp.br
[3] Email: adenilso@icmc.usp.br
[4] Email: jcmaldon@icmc.usp.br

doi:10.1016/j.entcs.2007.08.025

way the model is written impacts on how test inputs can be generated. Moreover, the decision of whether the implementation has already been tested enough depends on one's ability to determine how many undiscovered faults may remain in it. Usually the purpose of testing is not to demonstrate that the implementation is equivalent to its specification, since this goal is infeasible for most practical applications. Instead, this ideal equivalence is relaxed into a conformance relation [13,15]. The so-called conformance testing aims at demonstrating that the implementation behavior conforms (in some sense) to the behavior dictated by the specification [29].

The problem of generating test cases for conformance testing based on Finite State Machines (FSMs) has already been investigated [7,21,28,8,14,12]. However, there are many situations in which the modeling of the system as a FSM is cumbersome, due to the state explosion problem, or even impossible, due to the fact that there are some relevant aspects that can not be properly expressed, e.g., the passage of time. Some extensions to the FSM model have been proposed in order to overcome these problems [33,6,1]. Other extensions incorporate notions like context variables and input/output parameters, allowing the succinct representation of many different configurations [27]. Still others incorporate notions of time, allowing the model to capture the evolution of time [24,4,36].

An Extended Finite State Machine (EFSM) can be thought of as a folded FSM [27]. Given an EFSM, and assuming that domains are finite, it is possible to unfold it into a pure FSM by expanding the values of its parameters and variables. The resulting FSM can be used with FSM-based methods for test derivation with complete fault coverage, which means that all fault possibilities can be exhausted. Nonetheless, in most practical situations, this approach is unfeasible, mainly due to the state explosion effect [22,27].

Time plays an important role in determining the acceptability of system behavior in many system categories since not only the input/output relationship can be relevant, but also the period of time when those events occur may be important. In such cases, it is mandatory to be able to represent time constraints of the system, and to test whether a given implementation conforms to these constraints. There are some formalisms that allow the representation of various time related concepts, such as Timed Petri Nets [19] and Timed Automata [2,1,32,11]. Nonetheless, there are few, if any, methods that allow a satisfactory derivation of adequate test cases from those models.

We are interested in model-based methods for testing systems with time constraints. In particular, we are addressing the problem posed in tasks (i)-(iii) alluded to above, namely the construction of an adequate formalism for modeling systems and the automatic generation of test cases, as well as the determination of the expected outputs. These tasks are closely related, and should be considered together.

To this end, we define Timed EFSMs [5], or TEFSMs, by including the notion of explicit and implicit time advancement in the EFSM formalism. Then, we can adapt some well-established results, derived for FSMs and EFSMs, to the context of systems that require time constraints. In particular, we address the problem of configuration confirmation for TEFSMs in the same vein as done by Petrenko

et al. for EFSMs [27]. In that work, it is shown how the problem of configuration confirmation for EFSMs can be reduced to the problem of finding a path in an EFSM product. By defining a property that states when no such a path exists, model-checking techniques can be used to generate a confirming sequence. We show how the notion of product machines and confirming sequences can be applied to the extended formalism of TEFSMs. Given a configuration and a set of suspicious configurations, a confirming sequence is a sequence of (parameterized) inputs that allows us to distinguish the given configuration from suspicious configurations by comparing outputs and, possibly, observing the time indicated in each of the outputs. Finding a confirming sequence can also be seen as an extension of the state identification problem [20,16].

This paper is organized as follows. In Section 2 we present the concepts of EFSMs and Extended Timed Transition Systems [7]. In Section 3 we introduce the Timed Extended FSMs. The product of TEFSMs is presented in Section 4. In Section 5 we describe how the TEFSM product can be used in a model-checking set-up, and illustrate this process with a simple example in Section 6. Finally, in Section 7, we draw some concluding remarks and indicate possible directions for future research.

2 Basic Formal Concepts

In this section, we give a brief overview of the formal concepts that are involved in this work. First, we present EFSMs which are used to specify system requirements. Next, important aspects of extended timed transition systems are introduced.

2.1 Extended FSM Model

An EFSM is an extension of a conventional FSM. In contrast to FSMs, in the EFSM model we have to consider other items [27], such as input and output parameters, and context variables. Also, update and output functions, as well as predicates are defined over context variables and input parameters.

Let X and Y be finite sets of input and output symbols. Let R be a finite set of parameter symbols. For $z \in X \cup Y$, we denote by $R_z \subseteq R$ the set of parameters associated with z. Also D_z denotes the set of parameter valuations associated with z. An element of D_z maps R_z to some valuation domain. Similarly, let V be a finite set of context variable names, with D_V denoting a set of valuations for V. At this point, there is no need to further specify the valuation domains. An EFSM M over X, Y, R, V and the associated valuation domains is a tuple (S, T, s_0, λ_0), where S and T are finite sets of states and transitions, respectively, $s_0 \in S$ is the initial state, and λ_0 is an initial context variable valuation. Each transition $t \in T$ is a tuple (s, x, P, op, y, up, s'), where:

- $s, s' \in S$ are the source and the target states of the transition, respectively;
- $x \in X$ is the input symbol of the transition;
- $y \in Y$ is the output symbol of the transition;

- P, op and up are functions defined over valuations of the input parameters and context variables V, thus:
  - $P : D_x \times D_V \to \{True, False\}$ is the predicate of the transition;
  - $op : D_x \times D_V \to D_y$ is the output parameter function of the transition;
  - $up : D_x \times D_V \to D_V$ is the context update function of the transition.

Given an input x and the set of input parameter valuations D_x, a parameterized input is a pair (x, p_x), where $p_x \in D_x$. The parameterized outputs are defined in a similar way. A configuration of M is a pair $(s, \lambda) \in S \times D_V$, where s is a state and λ is a context variable valuation. A transition (s, x, P, op, y, up, s') is enabled for a configuration (s, λ) and parameterized input (x, p_x) if $P(p_x, \lambda)$ evaluates to true.

The machine starts from the initial configuration and operates as follows. Upon receiving an input along with the corresponding parameter valuation, and computes the predicates that are satisfied for the current configuration. From among the presently enabled transitions one will fire. By executing the chosen transition, the machine produces an output along with an output parameter valuation using of the output parameter function. The latter is computed by the output parameter valuation. The machine updates the current context variable valuation according to the context update function, and moves from the source to the target state of the transition.

An EFSM, furthermore, is considered to be:

- Predicate complete: for each pair $(s, x) \in S \times X$, every element in $D_x \times D_V$ evaluates at least one predicate to true among the set of all predicates guarding transitions leaving s with input x;
- Input complete: for each pair $(s, x) \in S \times X$, there exists at least one transition leaving state s with input x;
- Deterministic: any two transitions leaving the same state and with the same input have mutually exclusive predicates;
- Observable: for each state s and each input x, every outgoing transition from s on x has a distinct output symbol.

2.2 Extended Timed Transition Systems

We can extend the original *timed transition system* (TTS) notion of [7] by associating a set of clocks and invariant conditions with each state. All clocks in the model increase in an uniform way, according to a global time frame [1,2], and the corresponding invariant condition must hold in the current state of the model.

First, we say how clocks behave during system evolution [1]. Let C be the set of clock names (or clocks, for short), $\Phi(C)$ is the set of clock constraints δ in the form,

$$\delta := c \leq \tau \mid \tau \leq c \mid \neg\delta \mid \delta_1 \wedge \delta_2,$$

where c is a clock and $\tau \in \mathbb{Q}$ [5] is a time instant. A clock interpretation, ν, is a

[5] $\mathbb{Q}$ is the set of rationals and $\mathbb{Q}^{>0}$ is the set of positive rationals.

mapping from C to $\mathbb{Q}$. The set of clock interpretations is denoted by $[C \mapsto \mathbb{Q}]$. An interpretation ν over C satisfies $\delta \in \Phi(C)$, written $\nu \vDash \delta$, iff δ evaluates to true when each clock c is substituted by $\nu(c)$ in δ.

Let $\nu \in [C \mapsto \mathbb{Q}]$ be a clock interpretation. For $\tau \in \mathbb{Q}$, we define the clock interpretation $\nu + \tau$, which maps each clock c to the value $\nu(c) + \tau$. Also, for $K \subseteq C$, $[K \mapsto \tau]\nu$ is the clock interpretation that assigns $\tau \in \mathbb{Q}$ to each clock $c \in K$ and agrees with ν on the rest of the clocks.

An Extended TTS (ETTS) is given by a tuple $(S, s_0, X, C, Inv, \longrightarrow)$, where S is a finite set of states, $s_0 \in S$ is the initial state, X is a finite set of events, C is a finite set of clocks, $Inv : S \to \Phi(C)$ maps states to invariant conditions, and $\longrightarrow$ is a transition relation, where $\longrightarrow \subseteq (S \times X \times 2^C \times \Phi(C) \times S)$. A configuration is given by a pair (s, ν), where s is a state and ν is a clock interpretation. The initial configuration is given by (s_0, ν_0), where $\nu_0(c) = 0$, for all $c \in C$, is the initial clock interpretation, and $\nu_0 \vDash Inv(s_0)$. Given a configuration (s, ν), a transition (s, x, K, δ, s') indicates that from state s, receiving the input event x, and provided that ν satisfies δ, the system may move to state s', resetting all the clocks in K to zero. The ETTS always starts in the initial configuration (s_0, ν_0), and with the (global) time set to zero.

A time sequence is a sequence $\bar{\tau} = \tau_0\tau_1\tau_2\ldots$, where $\tau_i \in \mathbb{Q}$, $i \geq 0$, $\tau_0 = 0$, and $\tau_i \geq \tau_{i-1}$, $i \geq 1$. A timed sequence is a pair $(\bar{x}, \bar{\tau})$, where $\bar{\tau}$ is a time sequence and $\bar{x} = x_0x_1x_2\ldots$ is a sequence of input symbols. The intuitive idea is that the symbol x_i occurs at time τ_i. Given two configurations, (s_1, ν_1) and (s_2, ν_2), a time delay $\tau \geq 0$ and an input x, we say that (s_2, ν_2) evolves from (s_1, ν_1) over τ and x, denoted by $(s_1, \nu_1) \xrightarrow[\tau]{x} (s_2, \nu_2)$, iff there is a transition (s_1, x, K, δ, s_2) such that:

(i) $\nu_1 + \eta \vDash Inv(s_1)$ for all $0 \leq \eta \leq \tau$,

(ii) $\nu_1 + \tau \vDash \delta$,

(iii) $\nu_2 = [K \mapsto 0](\nu_1 + \tau)$, and

(iv) $\nu_2 \vDash Inv(s_2)$.

A sequence of configurations $\bar{\gamma} = \gamma_0\gamma_1\gamma_2\ldots$ is a run of M iff γ_0 is the initial configuration of M, and there is a timed input $(\bar{x}, \bar{\tau})$ such that

$$\gamma_{i-1} \xrightarrow[\theta_i]{x_i} \gamma_i, \text{ where } \theta_i = \tau_i - \tau_{i-1},\ i \geq 1.$$

In this case, we say that $\bar{\gamma}$ is a run of M over $(\bar{x}, \bar{\tau})$ from γ_0.

Note that, in a timed sequence $(\bar{x}, \bar{\tau})$, time evolves by $(\tau_i - \tau_{i-1})$ units from the moment when x_{i-1} occurred until x_i occurs (for $i > 1$). Intuitively a run captures the system evolution, as follows:

(i) it starts at state s_0, with all clocks set to zero;

(ii) time evolves by $\tau_1 - \tau_0 = \tau_1$ units;

(iii) at instant τ_1 the system changes to state s_1 on input x_1 while resetting clocks in K_1 to zero;

(iv) time evolves by another $\tau_2 - \tau_1$ units;

(v) at instant $\tau_1 + (\tau_2 - \tau_1) = \tau_2$ the system changes to state s_2 on input x_2 while resetting clocks in K_2 to zero;

(vi) and so on.

We can see that:

- a change of state can only occur when the transition (s, x, K, δ, s') is enabled, i.e., when δ is satisfied in the present configuration;
- clocks can be reset to zero in any transition;
- any clock reading is the elapsed time since the last instant it was reset to zero; and
- all clocks increase uniformly according to a global time frame.

3 The Timed EFSM model

In the previous sections, we have presented two formalisms: EFSMs and ETTSs. While EFSMs capture the relationships between inputs, outputs and context variables, ETTSs offer a treatment of time evolution and its constraints. We observe that there are several methods and techniques for deriving tests from (E)FSM models (e.g., [27,17,8,26]). However, the derivation of test cases from (E)TTSs is less established, although some works have considered it (e.g., [30,7,18]). It is worth combining both ETTSs and EFSMs formalisms in order to benefit from the power of both models in terms of expressiveness. This section redefines the EFSM model in order to capture real-time. We use the ETTS definition as inspiration for this purpose.

3.1 Creating a TEFSM model from an ETTS and an EFSM model

Let X be a finite set of inputs, Y be a finite set of outputs, C be a finite set of clocks, R be a finite set of parameters, and V be a finite set of context variables. A Timed Extended Finite State Machine, or TEFSM, M over X, Y, R, V, C, and the associated valuation domains is a tuple $(S, T, Inv, s_0, \nu_0, \lambda_0)$, where S and T are finite sets of states and transitions, respectively, Inv is a finite set of invariant conditions associated with states and, $s_0 \in S$ is the initial state, $\nu_0 = [C \mapsto 0]$ is the initial clock interpretation and λ_0 is an initial context variable valuation. In the TEFSM model: (i) the dynamic behavior is given by clocks and their resetting, as in the ETTS model; and (ii) the data and control flow are given by parameters and context variables, as in the EFSM model. A transition $t \in T$ is expressed by a tuple $(s, x, Q, K, op, y, up, s')$, where:

- s, x, s' and K are as defined in the ETTS formalism; see Section 2.2;
- op, y, and up are as defined in the EFSM formalism; see Section 2.1;
- $Q : D_x \times [C \mapsto \mathbb{Q}] \times D_V \to \{True, False\}$ is the predicate of the transition.

It can be seen that the TEFSM model comprises the EFSM formalism. That is, given a EFSM M over X, Y, R, V and some valuation domains, as defined in Section 2.1, we can construct a TEFSM model $\hat{M}$ over the same sets X, Y, R, V and the corresponding domains, by letting the clock set C be simply $\{c\}$. For each transition $t = (s, x, P, op, y, up, s')$ in M, we define a transition $\hat{t} = (s, x, Q, K, op, y, up, s')$ in $\hat{M}$ by letting $Q(p_x, \nu, \lambda) = P(p_x, \lambda)$, for any (p_x, ν, λ) in $D_x \times [C \mapsto \mathbb{Q}] \times D_V$. We also let $K = \emptyset$. Clearly, for any $p_x \in D_x$, $\lambda_v \in D_V$ and any clock interpretation $\nu \in [C \mapsto \mathbb{Q}]$, we have that $Q(p_x, \nu, \lambda_v)$ is true iff $P(p_x, \lambda_v)$ is true. For each state $s \in S$ in M, we define the invariant condition $\hat{Inv}(s) = (c \geq 0)$ in $\hat{M}$. Clearly, $\nu \vDash \hat{Inv}(s)$ for any $\nu \in [C \mapsto \mathbb{Q}]$ and $s \in S$.

Also, any ETTS model can be cast as a TEFSM model. For that, let $M = (S, s_0, X, C, Inv, \longrightarrow)$ be an ETTS model. Take a trivial common domain $\{0\}$ for all parameters and context variables, a single output symbol $Y = \{o\}$ and a single context variable $V = \{v\}$. For each parameter $z \in X \cup Y$, we define $R_z = \{z\}$. Then, the set of z-valuations is the singleton $D_z = \{p_z\}$, where p_z maps z to 0, for all $z \in X \cup \{o\}$. Similarly, $D_V = \{\lambda_v\}$, where λ_v maps v to 0. Now, a transition $t = (s, x, K, \delta, s')$ in M gives rise to a corresponding transition $\hat{t} = (s, x, Q, K, op, o, up, s')$ in $\hat{M}$, where:

- op maps (p_x, λ_v) to p_o;
- up maps (p_x, λ_v) to λ_v; and
- Q maps (p_x, ν, λ_v) to $True$ iff $\nu \vDash \delta$.

Here, the set of invariant conditions Inv for M is the same for $\hat{M}$. The initial state s_0 in $\hat{M}$ is the same initial state s_0 from M.

A configuration of a TEFSM M is a triple (s, ν, λ), where s is a state, ν is a clock interpretation and λ is a context variable valuation. The initial configuration is (s_0, ν_0, λ_0), where s_0 is the initial state of M, ν_0 is the initial clock interpretation of M and λ_0 is an initial context variable valuation of M. A configuration (s, ν, λ) is valid iff $\nu \vDash Inv(s)$. Let $\Gamma \subseteq S \times [C \mapsto \mathbb{Q}] \times D_V$ be the set of configurations of M.

3.2 The Operational Semantics for TEFSM models

Considering the dynamic behavior of ETTS models and the data and control flow of EFSM models, we define the operational semantics of a TEFSM M as follows.

Definition 3.1 Let $\gamma_i = (s_i, \nu_i, \lambda_i) \in \Gamma$, $i = 1, 2$, be two configurations of M. There is an implicit move from γ_1 to γ_2 iff

(i) $s_1 = s_2$,

(ii) $\lambda_1 = \lambda_2$,

(iii) $\nu_2 = \nu_1 + \tau$, for some $\tau \in \mathbb{Q}^{>0}$, and

(iv) $\nu_2 + \eta \vDash Inv(s_1)$, for all η, $0 \leq \eta \leq \tau$.

We denote such an implicit move by $\gamma_1 \underset{\tau}{\longrightarrow} \gamma_2$.

Definition 3.2 Let $\gamma_i = (s_i, \nu_i, \lambda_i) \in \Gamma$, $i = 1, 2$, be two configurations of M. Let (x, p_x) be a parameterized input and (y, p_y) be a parameterized output. There is an explicit move from γ_1 to γ_2 over (x, p_x) and yielding (y, p_y) iff there is a transition $(s_1, x, Q, K, op, y, up, s_2)$ in T such that:

(i) $\nu_2 = [K \mapsto 0]\nu_1$,

(ii) $\nu_2 \vDash Inv(s_2)$,

(iii) Q maps (p_x, ν_1, λ_1) to $True$,

(iv) op maps (p_x, λ_1) to p_y, and

(v) up maps (p_x, λ_1) to λ_2.

We denote such an explicit move by $\gamma_1 \xrightarrow{\chi/\xi} \gamma_2$, where $\chi = (x, p_x)$ e $\xi = (y, p_y)$.

Definition 3.3 Let $\gamma_i = (s_i, \nu_i, \lambda_i) \in \Gamma$, $i = 1, 2, 3$; $\tau \in \mathbb{Q}^{>0}$, (x, p_x) a parameterized input and (y, p_y) a parameterized output. If $\gamma_1 \xrightarrow[\tau]{} \gamma_2$ and $\gamma_2 \xrightarrow{\chi/\xi} \gamma_3$, where $\chi = (x, p_x)$ e $\xi = (y, p_y)$, then we say that there is a move from γ_1 to γ_3 and indicate this by $\gamma_1 \xrightarrow[\tau]{\chi/\xi} \gamma_3$.

Some of the decorations over and under $\longrightarrow$ may be dropped if they are clear from the context.

A parameterized input sequence is any sequence $\bar{\rho} = \rho_1\rho_2\ldots$ where each ρ_i is a parameterized input. A parameterized timed input sequence, or timed input, is a pair $(\bar{\rho}, \bar{\tau})$ where $\bar{\rho}$ is a parameterized input and $\bar{\tau}$ is a time sequence. Similar definitions hold for parameterized outputs. In particular a timed output is a parameterized timed output sequence.

A sequence of configurations $\bar{\gamma} = \gamma_0\gamma_1\gamma_2\ldots$ is a run of M iff there are a timed input $(\bar{\rho}, \bar{\tau})$ and a parameterized output sequence $\bar{\mu}$ such that

$$\gamma_{i-1} \xrightarrow[\theta_i]{\rho_i/\mu_i} \gamma_i, \text{ where } \theta_i = \tau_i - \tau_{i-1}, \text{ for all } i \geq 1.$$

We say that the run is over the timed input $(\bar{\rho}, \bar{\tau})$ and produces the timed output $(\bar{\mu}, \bar{\tau})$. We also say that $(\bar{\mu}, \bar{\tau})$, or $\bar{\mu}$, is produced by M from γ_0 in response to $(\bar{\rho}, \bar{\tau})$.

Some notions from the EFSM and ETTS models are extended to the TEFSM model:

- A TEFSM M is said to be predicate complete if, from any configuration (s, ν, λ) and given any parameterized input (x, p), there is a delay τ and a transition $(s, x, Q, K, op, y, up, s')$ such that Q evaluates $(p, \nu + \tau, \lambda)$ to $True$ and $\nu + \eta \vDash Inv(s)$, for all $0 \leq \eta \leq \tau$.
- The TEFSM M is complete if, for each state s there is a transition leaving s on any input symbol x.
- We say M is deterministic if, for any configuration (s, ν, λ), any parameterized input (x, p), and any time instant τ, there are no two different transitions $(s, x, Q_1, K_1, op_1, y_1, up_1, s_1)$ and $(s, x, Q_2, K_2, op_2, y_2, up_2, s_2)$ such that both Q_1

and Q_2 evaluate $(p, \nu + \tau, \lambda)$ to $True$.

- And, we say M is observable if, for any configuration (p, ν, λ), any parameterized input (x, p) there are no two transitions $(s, x, Q_1, K_1, op_1, y_1, up_1, s_1)$ and $(s, x, Q_2, K_2, op_2, y_2, up_2, s_2)$ with $y_1 \neq y_2$ and with Q_1 and Q_2 both evaluating (p, ν, λ) to $True$.

3.3 Configuration Distinguishability in the TEFSM model

Distinguishability of configurations in the Timed Extended Finite State Machine model is defined over parameterized input sequences. Two configurations γ and γ' of two distinct machines M and M', respectively, are distinguishable over a timed input $(\bar{\rho}, \bar{\tau})$ if the corresponding timed outputs $(\bar{\mu}, \bar{\tau})$ and $(\bar{\mu}', \bar{\tau}')$, produced by M and M' over $(\bar{\rho}, \bar{\tau})$ from γ and γ', respectively, are not compatible, in a sense to be defined shortly. We also say that $(\bar{\rho}, \bar{\tau})$ is a timed input separating those two configurations. We formalize these notions in the sequel, extending the definitions in [27]. Given a context variable valuation λ and a set of variables U, the U-projection of λ is the valuation obtained from λ by retaining the variables that are in the set U, denoted by $\lambda \downarrow U$. Similarly, for input symbols and their valuations, and for output symbols and the corresponding valuations.

Definition 3.4 Let y and y' be outputs of TEFSMs M and M', respectively. Let R and R' be the sets of parameters associated, respectively, with y and y'. The parameterized outputs (y, p) and (y', p') are said to be compatible if $y = y'$ and $p \downarrow R' = p' \downarrow R$. Two parameterized output sequences, $(y_1, p_1) \ldots (y_k, p_k)$ of M and $(y'_1, p'_1) \ldots (y'_k, p'_k)$ of M' are compatible if, for all $i = 1, \ldots, k$, the parameterized outputs (y_i, p_i) and (y'_i, p'_i) are compatible.

Intuitively, parameterized outputs are compatible when the output symbol is the same, and the output valuation agrees on all common output symbols. Distinguishability of configurations is defined as follows.

Definition 3.5 Given a timed input $\bar{\alpha} = (\bar{\rho}, \bar{\tau})$, a configuration γ of M and a configuration γ' of M' are distinguishable by $\bar{\alpha}$ if parameterized output sequence produced by M from γ in response to $\bar{\alpha}$ is not compatible with any parameterized output sequence that can be produced by M' from γ' in response to $\bar{\alpha}$. The timed input $\bar{\alpha}$ is said to be a sequence separating γ from γ'.

4 Timed Extended FSM Product

In Section 5 we extend to TEFSMs the method for the derivation of configuration confirming sequences defined in [27]. Since this method requires the notion of product machines, in this section we present the necessary extension of that notion to TEFSMs.

In the product of TEFSMs, the occurrence of implicit transitions can be ignored, since the global time frame which is used for all clock variables is the same for both

TEFSMs. This guarantees that the system evolution is maintained during implicit transitions.

Let $M^i = (S^i, Inv^i, T^i)$, $i = 1, 2$, and $\gamma^i = (s_0^i, \nu_0^i, \lambda_0^i)$, $i = 1, 2$, be two TEFSMs and their corresponding initial configurations. The product machine is denoted by $M^1 \times M^2$. We will use superscript 1 to denote elements of M^1, like R^1 is the set of parameters for M^1. Likewise, superscript 2 will indicate objects associated with M^2, like V^2 is the set of context variables of M^2. The superscript $1, 2$ is reserved for the product machine $M^1 \times M^2$.

The set of input symbols of $M^{1,2}$ is $X^{1,2} = X^1 \cup X^2$. Likewise, $Y^{1,2} = Y^1 \cup Y^2$. The set of parameters of $M^{1,2}$ is given by $R^{1,2} = R^1 \cup R^2$, with the proviso that for all $z \in R^1 \cap R^2$, the valuations of z in M^1 and M^2 have a common domain. It is clear that we are using the same parameter domains in $M^{1,2}$ as they were in M^1 and M^2. For any $z \in X^{1,2} \cup Y^{1,2}$, we let $R_z^{1,2} = R_z^1 \cup R_z^2$. Note that, given a valuation $r_z^{1,2}$ for elements in $R_z^{1,2}$ we can get valuations $r_z^1 = r_z^{1,2} \downarrow R_z^1$ and $r_z^2 = r_z^{1,2} \downarrow R_z^2$, for machines M^1 and M^2, respectively, and, moreover, $r_z^{1,2} = r_z^1 \cup r_z^2$. Similarly for clock interpretations and context variable valuations. We assume that clocks and context variables are disjoint, i.e., $C^{1,2} = C^1 \cup C^2$, with $C^1 \cap C^2 = \emptyset$, and $V^{1,2} = V^1 \cup V^2$, with $V^1 \cap V^2 = \emptyset$. As for the valuation domains, they are the same as in M^1 as in M^2. The set of states of $M^{1,2}$ is given by $S^{1,2} = S^1 \times (S^2 \cup \{fail\})$, where *fail* is a new state. The set of invariant conditions $Inv^{1,2}$ of $M^{1,2}$ maps $S^{1,2}$ to $\Phi(C^{1,2})$, and it is given by $Inv^{1,2}(s_1, s_2) = Inv^1(s_1) \wedge Inv^2(s_2)$, for all $(s_1, s_2) \in S^{1,2}$. Moreover, $Inv^{1,2}(s_1, fail) = Inv^1(s_1)$, for all $s_1 \in S^1$.

The initial configuration of $M^{1,2}$ will be given by $\gamma_0^{1,2} = ((s_0^1, s_0^2), (\nu_0^{1,2}, \lambda_0^{1,2}))$, where $\nu_0^{1,2} = \nu_0^1 \cup \nu_0^2$ and $\lambda_0^{1,2} = \lambda_0^1 \cup \lambda_0^2$. Note that we can take unions here, since clock and context variables are disjoint in M^1 and M^2.

It remains to specify the transitions of $M^{1,2}$. Let $(s_1^i, x, Q^i, K^i, op^i, y^i, up^i, s_2^i)$, $i = 1, 2$, be transitions of M^1 and M^2, both with the same input x. In the following definition we will be considering a parameterized input $(x, p_x^{1,2})$, a clock interpretation $\nu^{1,2}$ and a context variable valuation $\lambda^{1,2}$, all for the machine $M^{1,2}$. We also let $p_x^1 = p_x^{1,2} \downarrow R_x^1$ and $p_x^2 = p_x^{1,2} \downarrow R_x^2$. Likewise, we let $\nu^1 = \nu^{1,2} \downarrow C^1$ and $\nu^2 = \nu^{1,2} \downarrow C^2$, and also $\lambda^1 = \lambda^{1,2} \downarrow V^1$ and $\lambda^2 = \lambda^{1,2} \downarrow V^2$. There are two cases:

case 1: $y^1 = y^2$ and $op^1(p, \lambda) \downarrow R_{1,2} = op^2(p, \lambda) \downarrow R_{1,2}$, for all $(p, \lambda) \in D_x \times D_V$ where $R_{1,2} = R_{y^1}^1 \cap R_{y^2}^2$. That is, the output symbol is the same and the output valuations of both transitions are the same on each common output parameter. We add two transitions to $T^{1,2}$,

(i) $((s_1^1, s_1^2), x, Q, K, op, y^1, up, (s_2^1, s_2^2))$, where:

(a) $Q(p_x^{1,2}, \nu^{1,2}, \lambda^{1,2}) = Q^1(p_x^1, \nu^1, \lambda^1) \wedge Q^2(p_x^2, \nu^2, \lambda^2)$

(b) $K = K^1 \cup K^2$

(c) $op(p_x^{1,2}, \lambda^{1,2}) = op^1(p_x^1, \lambda^1) \cup op^2(p_x^2, \lambda^2)$. Recall that op^1 and op^2 coincide on common output parameters and so we can safely take the union.

(d) $up(p_x^{1,2}, \lambda^{1,2}) = up^1(p_x^1, \lambda^1) \cup up^2(p_x^2, \lambda^2)$. Recall that $V^1 \cap V^2 = \emptyset$.

(ii) $((s_1^1, s_1^2), x, Q, K, op, y^1, up, (s_2^1, fail))$, where:

(a) $Q(p_x^{1,2}, \nu^{1,2}, \lambda^{1,2}) = Q^1(p_x^1, \nu^1, \lambda^1) \wedge (\neg Q^2(p_x^2, \nu^2, \lambda^2))$
(b) $K = K^1$
(c) $op(p_x^{1,2}, \lambda^{1,2}) = op^1(p_x^1, \lambda^1)$
(d) $up(p_x^{1,2}, \lambda^{1,2}) = up^1(p_x^1, \lambda^1)$

case 2: Else, when the output valuations or the output symbols do not match, we add the transition $((s_1^1, s_1^2), x, Q, K, op, y, up, (s_2^1, fail))$ to $T^{1,2}$, where:
(i) $Q(p_x^{1,2}, \nu^{1,2}, \lambda^{1,2}) = Q^1(p_x^1, \nu^1, \lambda^1)$
(ii) $K = K^1$
(iii) $op(p_x^{1,2}, \lambda^{1,2}) = op^1(p_x^1, \lambda^1)$
(iv) $up(p_x^{1,2}, \lambda^{1,2}) = up^1(p_x^1, \lambda^1)$

Moreover, if $(s_1^1, x, Q, K, op, y, up, s_2^1)$ is a transition of M^1, we add to $M^{1,2}$ the transition $((s_1^1, fail), x, Q, K, op, y, up, (s_2^1, fail))$.

Suppose that the product machine is in the state (s_1^1, s_1^2), and on input $(x, p_x^{1,2})$ we find that M^1, on state s_1^1, has a transition on input (s_1^1, p_x^1), where p_x^1 is the reduction of $p_x^{1,2}$ to the parameters associated with x in M^1. Similarly, M^2, on state s_1^2, has a transition on (x, p_x^2). Moreover, the output of these transitions agree on the output symbol y, and also on valuations of any common output parameter of y in M^1 and in M^2. In this situation, we would want the product machine $M^{1,2}$ to enact both transitions of M^1 and M^2, componentwise. For that: (i) the same clocks are reset; (ii) the output parameter valuations are copied from M^1 and M^2; and (iii) both context updates are also carried over to $M^{1,2}$. But we can only enable this action in $M^{1,2}$ if both transitions in M^1 and M^2 are enabled. This is case 1(i).

Otherwise, we consider the situation where the transition in M^1 is enabled, but the one in M^2 is not. Here, we follow case 1(ii), and make the product machine $M^{1,2}$ enact the behavior of M^1 using for that the first state component, while the second component is marked as *fail*, thereby ignoring the transition from M^2. Note that, in this scenario, M^1 might have taken its transition, while M^2 would be forbidden to do so, even when their external behavior would have been indistinguishable. After the second state component is set to *fail*, $M^{1,2}$ behaves essentially as M^1.

Finally, when the product machine is in state (s_1^1, s_1^2), and we are considering an input $(x, p_x^{1,2})$, and we have picked two transitions from M^1 and M^2, starting respectively at s_1^1 and s_1^2, and whose output symbols or output parameter valuations do not match as above, then we proceed as in case 2. This is similar to case 1(ii) in that the second state component in $M^{1,2}$ is marked as *fail*, and $M^{1,2}$ uses the first state component to behave as M^1, from this moment on.

Consider configurations $\gamma^i = (s^i, \nu^i, \lambda^i)$ of machine M^i, $i = 1, 2$. Let $\bar{\rho} = (\bar{x}, \bar{p_x})$ be a parameterized input sequence for $M^1 \times M^2$, and let $\bar{\alpha} = (\bar{\rho}, \bar{\tau})$ is a timed input for $M^1 \times M^2$. Note that, M^1 and M^2 can be the same machine with different initial configurations. We say that $\bar{\alpha}$ is a separating sequence for γ^1 and γ^2 iff there is a run $\bar{\gamma} = \gamma_0\gamma_1 \ldots$ of $M^{1,2}$ over $\bar{\alpha}$, where $\gamma_0 = ((s^1, s^2), \nu^1 \cup \nu^2, \lambda^1 \cup \lambda^2)$ and for some $i \geq 1$, γ_i is a configuration of $M^{1,2}$ whose state is $(s_j^1, fail)$ for some $s_j^1 \in S^1$.

The problem of determining a separating sequence for two configurations of a

given TEFSM M can be reduced to a reachability problem. The reachability analysis is tractable but hard for EFSMs [23]. Indeed, for TEFSMs it is intractable. This is due to the temporal aspect within the new model. Another difficulty is the combinatorial explosion in the number of states in product machines. Some approaches try to overcome this difficulty by relaxing their restrictions. Approximation algorithms are also used when doing reachability analysis. Other approaches adapted known algorithms in order to manipulate symbolic data structures [34,9,35].

Other simpler contexts [25,3] present algorithms to obtain separating sequences. We postulate that these ideas can be adapted and extended in order to obtain separating sequences in the TEFSM formalism. Such separating sequences would be the result of the test case generation procedure. Moreover, we have been working with the notion of automata discretization in order to overcome the problem of infinite time instants. In addition, it is possible to modify conventional algorithms to reduce the state space generated by the product machine. Another alternative to obtain tractability in a timed approach for finding separating sequences is through the use of suspicious configurations [5]. In this case, we can choose a set of suspicious states, representing a important class fault, based on the expertise of test designers and on assumptions of implementations faults, as seen in [13,31].

5 Test Generation

This section outlines the main concepts for test case generation. First, we present some discussion on the main rationale of conformance testing. Second, we discuss the notion of confirming configurations, and how it is applied. At last, we discuss deriving test sequences by model-checking for TEFSMs.

5.1 Conformance Testing

Conformance testing aims at determining whether an implementation behaves in accordance with a given specification [21,15]. In general, an implementation is regarded as a black box, of which only input/output interfaces are known. In this situation, to verify whether an implementation is in conformance to a specification usually requires an infinite set of test cases in order to exhaust all error possibilities in the implementation. To overcome this problem, one possibility is to define a set of test hypotheses in order to reduce the number of test cases to be considered [13]. Test hypotheses strike a balance between two conflicting aspects. On the one hand, test hypotheses must be defined to be restrictive enough to render the method feasible and tractable. On the other hand, these hypotheses must be as less restrictive as possible, in such a way to be applicable to the largest possible set of implementations.

Conformance testing is guided by a conformance relation between the implementation and the specification [13]. In order to decide whether an implementation is in conformance to a specification, we observe the implementation's outputs to some applied inputs. Considering real-time systems, it must be also verified whether an implementation when stimulated by inputs responds with the expected outputs

within an allowed time interval.

The problem of using a conformance relation is the number of test sequences which should be obtained in order to verify whether each possible implementation is in conformance to a given specification. This problem is worse for timed systems, where there are infinite time instants for a transition to occur. To overcome this problem, we also need to enforce certain hypotheses about the implementation, as discussed in Section 5. This set of hypotheses will reduce the number of possible faults to be considered over the implementation and will render the method feasible in practical cases.

Several methods employ identification sequences to generate test cases from models. An identification sequence has the property of determining the correctness of the configuration reached after some input sequence is taken. Identification sequences may be defined as characterization sets [8,13], as distinguishing sequences [17] or as confirming configuration sequences (CCSs) [27], depending on the model and the generation method. A CCS which are investigated in this paper is a sequence that can increase the confidence that the correct configuration has been reached in the implementation.

5.2 Configuration Confirming Sequences

A configuration confirming sequence (CCS) is a timed input that can be applied to the implementation in order to increase the confidence on its correctness. A CCS can be derived from the product of two machines, one being a specification and the other an undesirable configuration. However, unlike the FSM models where a finite set of undesirable configurations can be postulated, with EFSM models and TEFSM models it is not possible, or desirable, to determine all undesirable configurations. To overcome this problem, a finite set of suspicious configurations is considered [27]. A set of suspicious configurations is derived from the specification to model suspicious implementations which can potentially have faults, reflecting the test designer's assumptions about the implementation faults. The suspicious configurations are extracted from the specification using a set of test hypotheses based on the fault model (e.g., [13]) and relying on the test designer's expertise.

These hypotheses define equivalence classes of implementations that must be put under testing, and they are used to reduce the number of possible implementations that need to be considered. In this work, we assume the following test hypotheses:

(i) Specifications and suspicious implementations are modeled by TEFSMs;

(ii) The number of clocks in the specification must be less than or equal to the number of clocks in the suspicious implementations; and

(iii) The same alphabets are used in both specification and suspicious implementations.

Given a configuration and a suspicious configuration, deriving a CCS can be reduced to the problem of finding a path in the product of two distinct TEFSMs, or of the same core TEFSM with distinct initial configurations. Such a sought

path would run from the initial state to a fail state. If the fail state can not be reached, then the suspicious configuration is equivalent to the original configuration. However, if a fail state is reachable, the model-checking algorithm will produce a counter-example, as a sequence of transitions that leads to this fail state [10]. This sequence would make a test case for the suspicious configuration. However, it is still necessary to identify in which moment each transition was taken, as well as the valuation of the input parameters associated with each input symbol. Gathering of this information forms a set of test cases. The test case is then used to exercise a real implementation, and the outputs are compared with the outputs produced by the specification over the same data. If a disagreement is found between corresponding outputs, then a fault has been identified.

5.3 Model-checking

Design errors frequently occur when conventional simulation and testing techniques are used to check safety systems. Model checking is a set of techniques for the automatic analysis of reactive and real-time systems. Some model checking algorithms have been modified to discover such errors, thus providing more quality and accuracy in system verifications. In general, given a transition system and a property, the model checking problem is to decide whether the property holds or not in the model represented by the transition system. If not, the model checking algorithm provides a counterexample, i.e. an execution over the model that violates the property [23].

Reachability analysis is a special kind of model-checking method that can be applied in a formal model. In general, given a special state to be found in a model, the reachability analysis decides if it is possible to move from the initial state to the final special state.

To summarize, to automatically test implementations based on a specification represented by a machine M, the following steps are performed:

(i) An empty set TC of test cases is defined.

(ii) Given a configuration γ of M, a set of suspicious configurations Γ is defined, based on test hypotheses, fault models and some specific test engineer's objectives.

(iii) For each suspicious configuration $\gamma_s \in \Gamma$, the product of M with itself is constructed, having γ as the initial configuration of the first instance of M in the product, and having γ_s as the initial configuration of the second instance.

(iv) Reachability analysis is carried out, in order to find a path to a fail state in the product machine. If such a path is found, it is added to TC.

(v) For each $tc \in TC$, a time and an input parameter valuation sequences are derived so as to satisfy the predicates along the path specified by tc.

(vi) Each path in TC, with its associated data, is applied to the real implementation under testing.

6 An Example

We are given two TEFSMs M and N, where M is a specification and N is a suspicious implementation of M. We obtain the product of these machines, $M \times N$, by applying our method. In this example, as is usual in practice, N has the same transitions as M. They differ only in their associated initial configurations. Accordingly, we will denote the product by $M^0 \times M^1$, where M^0 is the specification and M^1 is the suspicious configuration. The TEFSM M depicted in Figure 1.

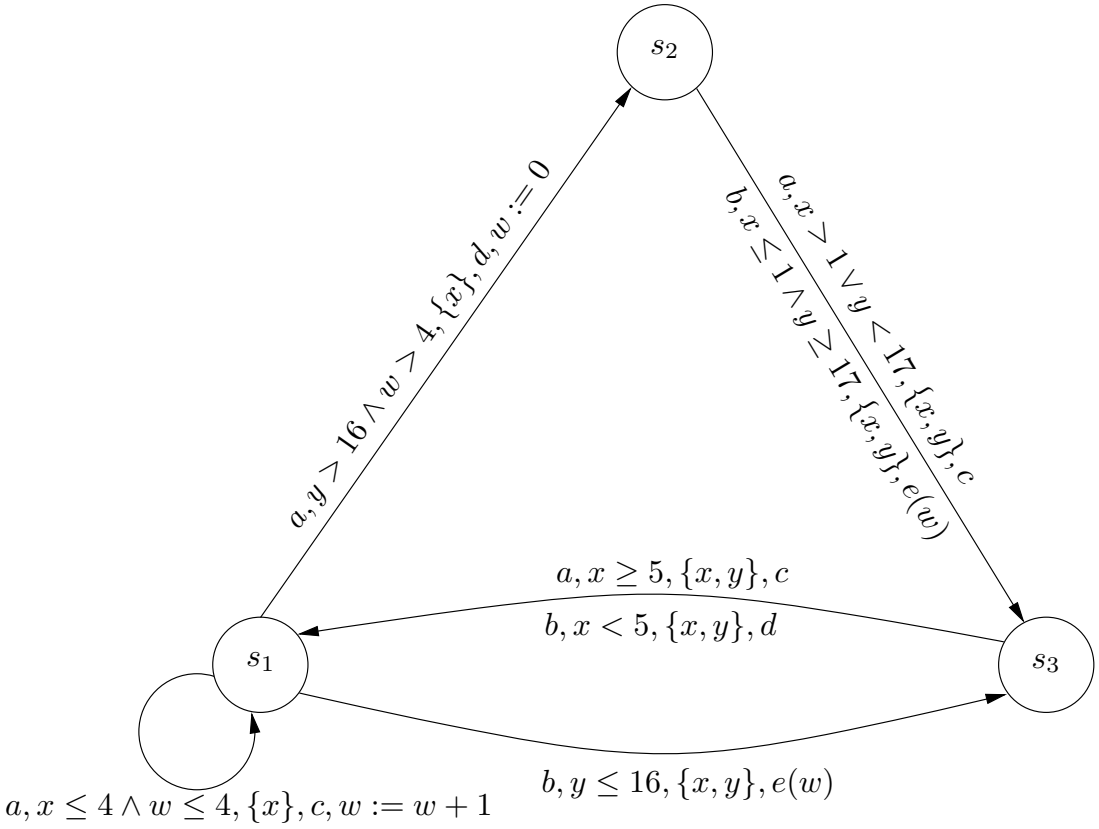


Fig. 1. The TEFSM M.

It has three states and seven transitions. The input set is $\{a, b\}$ and the output set is $\{c, d, e\}$. Furthermore, M has two clock variables, x and y, and one context variable w. There are no parameters associated with the input symbols, i.e. $R_a = R_b = \emptyset$. Likewise, $R_c = R_d = \emptyset$. For each state s in M, the control remains in s whenever its invariant condition is satisfied. The output e has only one associated parameter. In this case, it is not necessary to name the parameter. Instead, in Figure 1 and in the sequel we write $e(w)$ to indicate that the current value of the context variable w is to be attributed to the parameter associated with e. In the figure, each arrow is labeled by a sequence of items. The first three are always the input symbol, the predicate function and the set of clocks to be reset in the transition, respectively. Next, comes the output symbols, either c or d, and we write directly $e(w)$ to indicate both the output symbol and the value of its parameter. Finally, if the value of the context variable w is altered by the transition, this is indicated by the attribution that appears at the end of the label; if the value of w is not altered by the transition we simply omit the trivial expression $w := w$.

A configuration of M is given by a state, a clock interpretation and a context

variable valuation. Hence, a configuration of M will be denoted by $(s,(n,m),k)$ indicating that the machine is in state s, n and m are the values for the clock variables x and y, respectively, and k is the value for the context variable w. The integers are selected as a common valuation domain. In the configuration $(s_1,(3,2),4)$ the transition $a, x \leq 4 \wedge w \leq 4, \{x\}, c, w := w+1$, from s_1 to itself, is enabled. Likewise, the transition $b, y \leq 16, \{x,y\}, e(w)$, from s_1 to s_3, is also enabled.

For the product, let M^0 designate M with the initial configuration $(s_1,(0,0),2)$, and let M^1 designate M with initial configuration $(s_1,(4,2),5)$. The TEFSM product of $M^0 \times M^1$ is shown in Figure 2. To simplify the notation in the example, we will use subscript i to denote items of machine M^i, for $i = 0, 1$, e.g. x_1 represents the clock variable x of M_1, while w_0 denotes the variable w in M^0.

The initial configuration of $M^0 \times M^1$ is denoted by $((s_1,s_1),(0,0,2,4),(2,5))$, where we list first the items corresponding to M^0, followed by the items associated with M^1. Note that, in the figure, states are represented by subscripts, e.g., the state (s_1,s_1) in the product is named s_{11}.

By inspection of the product, we can see that the input b enables the transition to fire, since clock conditions on y_0 and y_1 are satisfied for the initial configuration. After that the transition is taken, the new configuration is given by $((s_3,s_3),(0,0,2,0),(0,5))$. It is easy to see that neither input a nor input b will enable transitions to fire so as to reach, directly, a fail state. Note that, every clock variable was reset to zero, and transition guards are excluding for clock variables x_0 and x_1. If the input b occurs within less than 5 time units, the configuration becomes $((s_1,s_1),(0,0,2,0),(0,5))$. Otherwise, if the time evolves for more than 5 time units, only the input a could stimulate the machine to change configurations. The new configuration would still be $((s_1,s_1),(0,0,2,0),(0,5))$. Both transitions would drive the control back to the initial state, where the transition stimulated by input b is the unique one enabled to fire. This cycle would be executed repeatedly and a fail state would not be reached.

Another possibility is to take the transition on the input a. It is easy to see that the input a separates the configurations $(0,0,2)$ from $(4,2,5)$. The final configuration reached is $((s_1,fail),(0,0,3,4),(2,5))$. On the other hand, the control can be kept within the state (s_1,s_1), by a continuous time evolution. After that, the stimulation by input a enables the transition to fire, and the configuration $((s_1,s_1),(1,1,2,5),(3,5))$ can be reached. Then, the transition from state (s_1,s_1), on input a and with associated predicate $x_0 \leq 4 \wedge w_0 \leq 4 \wedge (w_1 > 4 \vee x_1 > 4)$ is enabled and takes the machine to the fail state $(s_1,fail)$. Here, only the clock variable x_0 is reset, and the context variable w_0 is updated by one unit. The new configuration will be $((s_1,fail),(0,1,3,5),(3,5))$. From here, we see that input a separates the configuration $(0,0,2)$ from $(4,2,5)$, after some time passes. The new configuration that can be reached in this case is $((s_1,fail),(0,1,3,5),(3,5))$.

If we consider another situation, where the initial configurations of M^0 and M^1, respectively, are given by $(0,0,2)$ and $(4,2,4)$, another run of $M^0 \times M^1$ will also reach the fail state. In this case, a reachability analysis shows that the fail state of $M^0 \times M^1$ can only be reached when a sequence of one or two consecutive inputs a

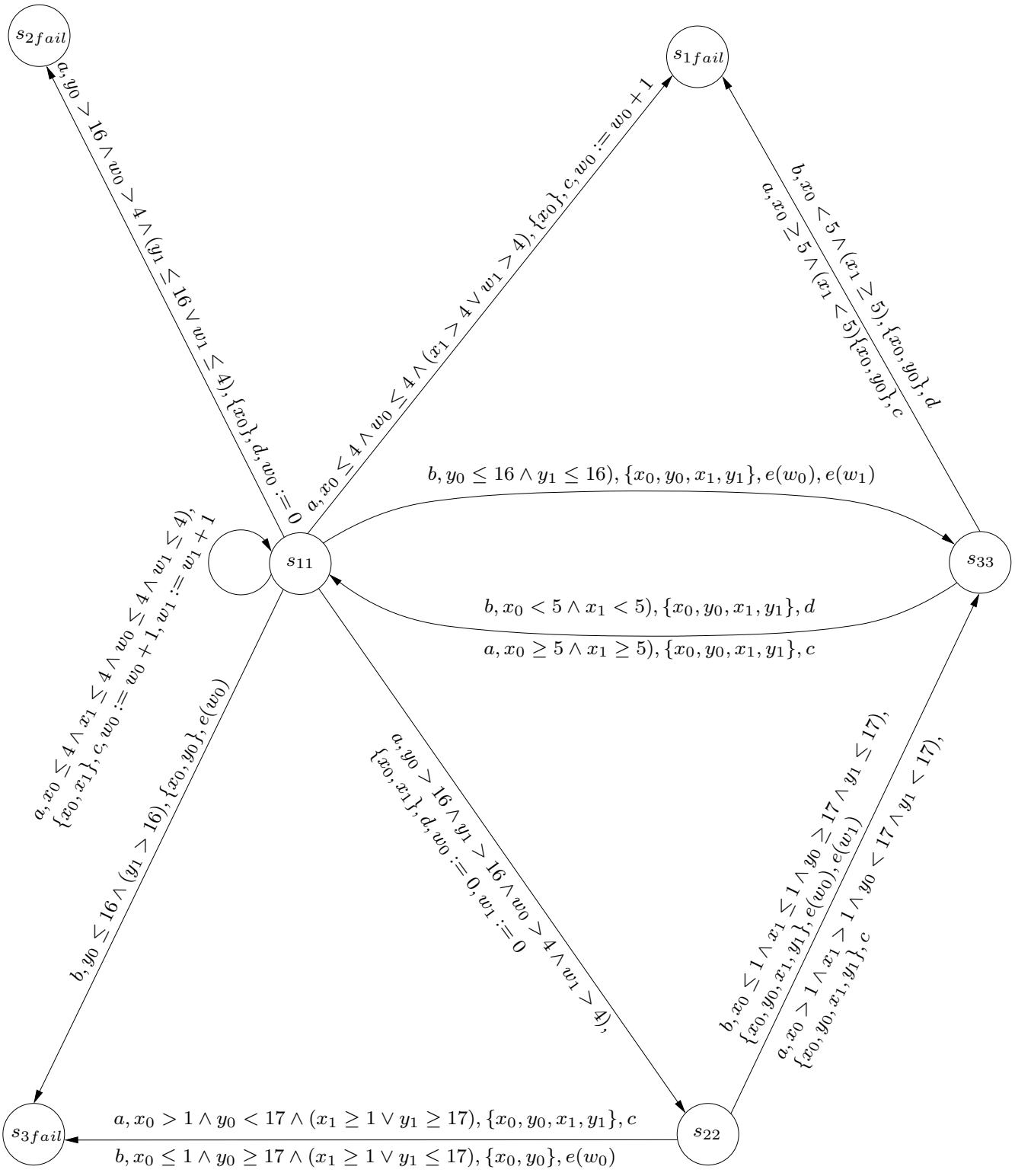


Fig. 2. The TEFSM product of M with itself.

is applied.

In the example, M^0 represents the specification, M^1 represents a suspicious implementation, and the product $M^0 \times M^1$ is used to find sequences of configurations that show non conformance between a suspicious implementation and the specification. We can derive traces from the reachability analysis of $M^0 \times M^1$. The resulting traces are runs that reach the fail state in the product machine, starting from the initial configurations of the participating TEFSMs.

7 Concluding Remarks

The ability to derive test cases from formal models opens the possibility that we can construct more rigorous and dependable systems, by providing a sound basis for the validation of the systems' behaviors. There is a direct relationship between the kinds of systems that a given model can deal with and the availability of methods for deriving test cases. The FSM and EFSM models are well-established and have been intensively investigated. One important feature they both lack is the ability to deal with time. In this paper we define TEFSMs as a model that extends the EFSM model with the notion of time. From that, we discussed an extended method for deriving configuration confirming sequences for TEFSMs, a step toward automating the generation of test cases from these models.

Although we can argue that both the model and the generation method can be used, we do not have answers for pragmatic questions, such as (i) how difficult is it to describe a system using TEFSMs and (ii) how large are the models we can handle. To answer these questions, it is necessary to deepen the investigations and implement adequate supporting software tools. We are currently working in this direction.

Other aspects that can be investigated include how to allow time constraint to be defined over outputs. We note that our definition does not deal with constraints that may reflect output response that is not instantaneous. The input and output occur in the same time instant. We are considering how this extension might impact the test case generation methods.

References

[1] Alur, R., *Timed automata*, in: *CAV*, number 1633 in LNCS, 1999, pp. 8–22.

[2] Alur, R. and D. L. Dill, *A theory of timed automata*, Theoretical Computer Science **126** (1994), pp. 183–235.
URL citeseer.ist.psu.edu/alur94theory.html

[3] Alur, R., M. McDougall and Z. Yang, *Exploiting behavioral hierarchy for efficient model checking.*, in: *CAV*, 2002, pp. 338–342.

[4] Behrmann, G., K. G. Larsen, J. Pearson, C. Weise and W. Yi, *Efficient timed reachability analysis using clock difference diagrams*, in: *Computer Aided Verification*, 1999, pp. 341–353.
URL citeseer.ist.psu.edu/article/behrmann99efficient.html

[5] Bonifácio, A. L., A. V. Moura, A. d. S. Simão and J. C. Maldonado, *Conformance Testing by Model Checking Timed Extended Finite State Machines*, in: *Brazilian Symposium on Formal Methods (SBMF'06)*, Natal, 2006, pp. 43–58.

[6] Campos, S. V., M. Minea, W. Marrero, E. M. Clarke and H. Hiraishi, *Computing quantitative characteristics of finite-state real-time systems*, in: *Proc. 15th IEEE Real-Time Systems Symp.* (1994), pp. 266–270, san Juan, Porto Rico.

[7] Cardell-Oliver, R., *Conformance tests for real-time systems with timed automata specifications*, Formal Aspects of Computing **12** (2000), pp. 350–371.
URL citeseer.ist.psu.edu/385816.html

[8] Chow, T. S., *Testing software design modeled by finite-state machines*, IEEE Transactions on Software Engineering **4** (1978), pp. 178–187.

[9] Cimatti, A., E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, *Integrating bdd-based and sat-based symbolic model checking.*, in: *FroCos*, 2002, pp. 49–56.

[10] da Silva, D. A. and P. D. L. Machado, *Towards test purpose generation from ctl properties for reactive systems.*, Electr. Notes Theor. Comput. Sci. **164** (2006), pp. 29–40.

[11] Dickhöfer, M. and T. Wilke, *Timed alternating tree automata: the automata-theoretic solution to the tctl model checking problem*, in: *26th ICALP*, LNCS **1644**, 1999, pp. 281–290.
URL citeseer.ist.psu.edu/article/dickhfer99timed.html

[12] Dorofeeva, R., K. El-Fakih and N. Yevtushenko, *An improved conformance testing method*, in: *Formal Techniques for Networked and Distributed Systems*, Lecture Notes in Computer Science **3731** (2005), pp. 204–218.

[13] En-Nouaary, A., R. Dssouli and F. Khendek, *Timed wp-method: Testing real-time systems*, IEEE Trans. Softw. Eng. **28** (2002), pp. 1023–1038.

[14] Fujiwara, S., G. V. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, *Test selection based on finite state models*, IEEE Transaction on Software Engineering **17** (1991).

[15] Gargantini, A., *Conformance testing*, in: M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker and A. Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, Lecture Notes in Computer Science **3472** (2005), pp. 87–111.

[16] Gill, A., "Introduction to the theory of finite-state machines," McGraw-Hill, New York, 1962.

[17] Gonnenc, G., *A method for the design of fault detection experiments*, IEEE Transactions on Computing **19** (1970), pp. 551–558.

[18] Higashino, T., A. Nakata, K. Taniguchi and A. R. Cavalli, *Generating test cases for a timed i/o automaton model*, in: *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems* (1999), pp. 197–214.

[19] Hirai, T., *An application of temporal linear logic to Timed Petri Nets*, in: *Proceedings of the Petri Nets'99 Workshop on Applications of Petri Nets to Intelligent System Development*, 1999, pp. 2–13.

[20] Krichen, M., *State identification*, in: M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker and A. Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, Lecture Notes in Computer Science **3472** (2005), pp. 87–111.

[21] Krichen, M. and S. Tripakis, *Black-box conformance testing for real-time systems*, in: *Model Checking Software: 11th International SPIN Workshop*, number 2989 in Lecture Notes in Computer Science, Barcelona, Spain, 2004, pp. 109–126.

[22] McMillan, K. L., "Symbolic Model Checking: An Approach to the State Explosion Problem," Kluwer Academic, 1993.

[23] Merz, S., *Model checking: A tutorial overview*, in: F. C. et al., editor, *Modeling and Verification of Parallel Processes*, Lecture Notes in Computer Science **2067**, Springer-Verlag, Berlin, 2001 pp. 3–38.

[24] Møller, J. B., *Simplifying fixpoint computations in verification of real-time systems* (2002).
URL http://citeseer.ist.psu.edu/540135.html

[25] Nr, B., M. Dickhofer and T. Wilke, *The automata-theoretic method works for TCTL model checking* (1998).
URL http://citeseer.ist.psu.edu/44733.html

[26] Offutt, A. J., Y. Xiong and S. Liu, *Criteria for generating specification-based tests*, in: *Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, Las Vegas, NV, 1999, pp. 41–50.

[27] Petrenko, A., S. Boroday and R. Groz, *Confirming configurations in efsm testing*, IEEE Trans. Softw. Eng. **30** (2004), pp. 29–42.

[28] Petrenko, A. and N. Yevtushenko, *Testing from partial deterministic fsm specifications*, IEEE Transactions on Computers **54** (2005).

[29] Tretmans, J., *Test generation with inputs, outputs, and quiescence.*, in: T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, Lecture Notes in Computer Science **1055** (1996), pp. 127–146.

[30] Tretmans, J., *Testing concurrent systems: A formal approach*, in: J. Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, Lecture Notes in Computer Science **1664** (1999), pp. 46–65.

[31] Wang, C.-J. and M. T. Liu, *Generating test cases for efsm with given fault models.*, in: *INFOCOM*, 1993, pp. 774–781.

[32] Wang, F., *Efficient verification of timed automata with bdd-like data structures.*, STTT **6** (2004), pp. 77–97.

[33] Wang, F., *Formal verification of timed systems: A survey and perspective.*, Proceedings of the IEEE **92** (2004), pp. 1283–1307.

[34] Wang, F., *Symbolic parametric safety analysis of linear hybrid systems with bdd-like data-structures*, Software Engineering, IEEE Transactions on **31** (2005), pp. 38–51.

[35] Wang, F., *Under-approximation of the Greatest Fixpoints in Real-Time System Verification*, ArXiv Computer Science e-prints (2005), pp. 1060–+.

[36] Wang, F., G.-D. Hwang and F. Yu, *Tctl inevitability analysis of dense-time systems.*, in: *CIAA*, 2003, pp. 176–187.

Apêndice I

A. S. Simão, A. Petrenko, N. Yevtushenko. Generating Reduced Tests for FSMs with Extra States. In: 21st IFIP Int. Conference on Testing of Communicating Systems and the 9th Int. Workshop on Formal Approaches to Testing of Software, 2009. p. 129-147, Eindhoven, Holanda.

Generating Reduced Tests for FSMs with Extra States

Adenilso Simão[1,2], Alexandre Petrenko[2], and Nina Yevtushenko[3]

[1] São Paulo University, São Carlos, São Paulo, Brazil
[2] Centre de recherche informatique de Montreal (CRIM)
Montreal, Quebec, Canada
[3] Tomsk State University, Tomsk, Russia
adenilso@icmc.usp.br, petrenko@crim.ca,
ninayevtushenko@yahoo.com

Abstract. We address the problem of generating tests from a deterministic Finite State Machine to provide full fault coverage even if the faults may introduce extra states in the implementations. It is well-known that such tests should include the sequences in the so-called traversal set, which contains all sequences of length defined by the number of extra states. Therefore, the only apparent opportunity to produce shorter tests is to find within a test suite a suitable arrangement of the sequences in the inescapable traversal set. We observe that the direct concatenation of the traversal set to a given state cover, suggested by all existing generation methods with full fault coverage, results in extensive test branching, when a test has to be repeatedly executed to apply all the sequences of the traversal set. In this paper, we state conditions which allow distributing these sequences over several tests. We then utilize these conditions to elaborate a method, called SPY-method, which shortens tests by avoiding test branching as much as possible. We present the results of the experimental comparison of the proposed method with an existing method which indicate that the resulting save can be up to 40%.

1 Introduction

Finite State Machines (FSMs) have been used to model systems in many areas, such as hardware design, formal language recognition, conformance testing of protocols [1] and object-oriented software testing [2]. Regarding test generation, one of the main advantages of using FSMs is the existence of generation methods which guarantee full fault coverage: given a specification FSM with n states and any black-box implementation which can be modelled as an FSM with at most m states, $m \geq n$, the methods generate a test suite, often called m-complete test suite, which has the ability to detect all faults in any such implementations. In the particular case of $m = n$, there are many efficient methods which generate complete test suites [7] [3] [5] [10] [4].

However, on the other hand, in spite of the fact that the problem of generating m-complete test suites for $m > n$ is a longstanding one which can be traced back to the work of Moore [11] and Hennie [9], it has received much less attention compared to the problem of constructing n-complete test suites. One of the main reasons might be the fact that test generation becomes more challenging in the case of extra states. It is known that an m-complete test suite should include each sequence in the so-called

M. Núñez et al. (Eds.): TESTCOM/FATES 2009, LNCS 5826, pp. 129–145, 2009.

traversal set, which contains all input sequences with $m - n + 1$ inputs [13]. Moreover, the traversal set should be applied to each state of the specification. Not surprisingly, all, not numerous, existing methods for generating m-complete test suites [13] [3] [5] [14] [4] [8] [12] do exactly this and differ only in a type of state identification sequences they add to traversal sequences.

Driven by this observation and the obvious absence of significant progress in solving the longstanding problem of generating m-complete test suite, we revisit it in this paper and aim at answering the question whether m-complete test suite is irreducible due to the inevitability of the traversal set.

We observe that a considerable part of an m-complete test suite is not related to the traversal set itself, but to the test branching when a test has to be repeatedly executed to apply all the sequences of the traversal set. Apparently, the test length reduction can only be achieved by reducing the test branching, which in turn can be obtained by distributing the traversal set over several tests. The caveat is that an arbitrary distribution of the traversal set may break the m-completeness of a resulting test suite. Thus, we need first to establish conditions for a distribution of the traversal set such that the m-completeness of a test suite is preserved. The main idea developed in this paper is to distribute it among those tests in a test suite which are convergent, i.e., transfer to the same state, in all FSMs of the fault domain which pass the test suite. The approach we elaborate is based on properties of FSM tests, namely their convergence and divergence. We investigate when the convergence and divergence of tests in the specification (which can be easily checked) can be safely assumed to also hold in the implementation under test. The divergence of two tests can be witnessed by different outputs produced by the tests. On the other hand, although convergence of two tests cannot be directly ascertained by considering only the two tests, we show that the knowledge of the maximum number of states of FSMs in the fault domain can be used to formulate conditions for the convergence of tests. We then use the notion of convergence and divergence to state necessary and sufficient conditions for a test suite to be m-complete.

Based on these conditions, we elaborate a method, called SPY-method, for m-complete test suite generation. The method distributes the sequences of the traversal set over several tests in order to reduce test branching and generate shorter test suites. To assess the potential saving which can be obtained with the approach proposed in this paper, we experimentally compare it with the HSI method [14]. The results suggest that SPY-method can generate test suites up to 40% shorter, on average.

The rest of the paper is organized as follows. In Section 2, we provide the necessary basic definitions. In Section 3, we formally state the problem of generating m-complete test suites and discuss existing methods. In Section 4, we investigate test properties and formulate conditions for guaranteeing the m-completeness of test suites. In Section 5, we develop a generation method based on the proposed conditions. In Section 6, the method is illustrated on an example. Experimental results are reported in Section 7 and Section 8 concludes the paper.

2 Definitions

A Finite State Machine is a (complete) deterministic Mealy machine, which can be defined as follows.

Definition 1. *A* Finite State Machine (FSM) $\mathcal{S}$ *is a 6-tuple* $(S, s_0, X, Y, \delta_S, \lambda_S)$*, where*

- S *is a finite set of states with the initial state* s_0*,*
- X *is a finite set of inputs,*
- Y *is a finite set of outputs,*
- $\delta_S : S \times X \to S$ *is a transition function, and*
- $\lambda_S : S \times X \to Y$ *is an output function.*

A tuple $(s, x) \in S \times X$ is a *transition* of $\mathcal{S}$. We extend the transition and output functions from input symbols to input sequences, including the empty sequence ε, as usual: for $s \in S$, $\delta_S(s, \varepsilon) = s$ and $\lambda_S(s, \varepsilon) = \varepsilon$; and for input sequence α and input x, $\delta_S(s, \alpha x) = \delta_S(\delta_S(s, \alpha), x)$ and $\lambda_S(s, \alpha x) = \lambda_S(s, \alpha)\lambda_S(\delta_S(s, \alpha), x)$. An FSM $\mathcal{S}$ is said to be *initially connected*, if for each state $s \in S$, there exists an input sequence $\alpha \in X^*$, called a *transfer* sequence for state s, such that $\delta_S(s_0, \alpha) = s$. In this paper, only initially connected machines are considered. Input sequences *converge* if they are transfer sequences for the same state. Similarly, input sequences *diverge* if they are transfer sequences for different states of the same FSM. A set $K \subseteq X^*$ is a *state cover* for $\mathcal{S}$ if it contains at least one transfer sequence for each state of $\mathcal{S}$. A state cover is *minimal* if it contains exactly one transfer sequence for each state. A set $A \subseteq X^*$ *covers* a transition (s, x) if there exist $\alpha, \alpha x \in A$, where α is a transfer sequence for s. The set A is a *transition cover* for $\mathcal{S}$ if it covers every transition of $\mathcal{S}$. A set of sequences is *initialized*, if it contains the empty sequence.

Given sequences $\alpha, \beta, \gamma \in X^*$, if $\beta = \alpha\gamma$, then α is a *prefix* of β, and γ is a *suffix* of β; if γ is not the empty sequence, then α is a *proper* prefix of β. We also say that a prefix of γ *extends* α (*in* β) and that β is an *extension* of α. We denote by *pref*(β) the set of all prefixes of β. For a set of sequences A, *pref*(A) is the union of *pref*(β), for all $\beta \in A$. If A = *pref*(A), then we say that A is *prefix closed*. Given two sets of sequences A and B, we denote by $A.B$ the set of sequences $A.B = \{\alpha\beta \mid \alpha \in A \text{ and } \beta \in B\}$. We will slightly abuse the notation by writing $\alpha.B$ instead of $\{\alpha\}.B$ and $A.\beta$ instead of $A.\{\beta\}$. For a natural number k, we denote by $X^{\leq k}$ the set of all input sequences of length at most k.

Given a set $A \subseteq X^*$, states s and s' are *A-equivalent*, if $\lambda_S(s, \gamma) = \lambda_S(s', \gamma)$ for all $\gamma \in A$. Otherwise, s and s' are *A-distinguishable*. We say that γ distinguishes s and s', if s and s' are $\{\gamma\}$-distinguishable. States s, s' are *equivalent*, if they are X^*-equivalent. Similarly, they are *distinguishable* if they are X^*-distinguishable. We define distinguishability and equivalence of machines as a corresponding relation between their initial states. An FSM is *minimal*, if all states are pairwise distinguishable. In this paper, all the FSMs are assumed to be minimal. A *characterization set* is a set of sequences W such that every two states are W-distinguishable. The set $W_s \subseteq W$ is a *state identifier* for state s if any other state is W_s-distinguishable from s. A *family of harmonized state identifiers* is a collection of sets $\{H_s \mid s \in S\}$, such that states s and s' are (*pref*(H_s) $\cap$ *pref*($H_{s'}$))-distinguishable.

3 Problem Statement and Existing Methods

In this section, we discuss the problem of generating test suites with full fault coverage along with the existing methods and present the main idea of the approach elaborated in this paper.

Henceforth, we assume that $S = (S, s_0, X, Y, \delta_S, \lambda_S)$ and $Q = (Q, q_0, X, Y', \delta_Q, \lambda_Q)$ are a specification FSM and an implementation FSM, respectively. Moreover, n is the number of states of S. We denote by $\Im$ the set of all minimal implementation FSMs with the same input alphabet as S. The set $\Im$ is called a *fault domain* for S. For $m \geq n$, let $\Im_m$ be the FSMs of $\Im$ with at most m states, i.e., the set $\Im_m$ represents all faults that can occur in an implementation of S with at most m states. We denote the maximum number of extra states that an implementation may have by $\Delta = m - n$. Faults can be detected by tests, which are input sequences of the specification FSM S.

Definition 2. *An input sequence of FSM S is called a* test case *(or simply a* test*) of S. A* test suite *of S is a finite prefix closed set of tests of S. A test suite T of FSM S is* m-complete, *if for each FSM $Q \in \Im_m$, distinguishable from S, there exists a test in T that distinguishes them.*

An FSM *passes* a test suite T if it is T-equivalent to the specification. Thus, a test suite is m-complete if the FSMs in $\Im_m$ which pass it are equivalent to the specification. Two tests α and β in a given test suite T are *T-separable*, if there exist $\alpha\gamma, \beta\gamma \in T$, such that states $\delta_S(s_0, \alpha)$ and $\delta_S(s_0, \beta)$ are $\{\gamma\}$-distinguishable. Clearly, if T-separable tests α and β are convergent in some implementation FSM, it can be distinguished from S by either $\alpha\gamma$ or $\beta\gamma$.

Since the distinguishability of FSMs is defined as the corresponding relation of their initial states, tests are assumed to be applied in the initial state. For accounting to the reset operation required to bring the FSMs to the initial state, we define the length of a test α as $|\alpha| + 1$, where $|\alpha|$ is the number of input symbols in α. As the application of a test results in the application of all its prefixes, the length of a test suite T, denoted by $len(T)$, is the sum of the lengths of all tests in T which are not proper prefixes of other tests in T.

In this paper, we address the problem of generating an m-complete test suite, when implementation FSMs can have more that n states, i.e., $m \geq n$. This problem has received much less attention compared to the (classical) problem of constructing n-complete test suites, often called checking experiments. One of the main reasons might be the fact that test generation becomes more challenging. To illustrate this, let us consider the FSMs in Figures 1 and 2, where S_0 is the specification machine and S_1 is an implementation machine, which has two extra states. Notice that states 1 and 2 in S_1 are similar to states 1 and 2 in S_0, except that S_1 has two extra states 1' and 2', and the transition (2, *b*) leads to an "erroneous" state 2'.

The shortest test able to distinguish S_0 and S_1 should be formed by the input sequence *a*, which is a transfer sequence for state 2, and the input sequence *baa*. Indeed, for any other input sequence of length three, it is possible to construct a distinguishable FSM with two extra states for which only that particular sequence applied to a proper state distinguishes it from S_0. As those FSMs are in the fault

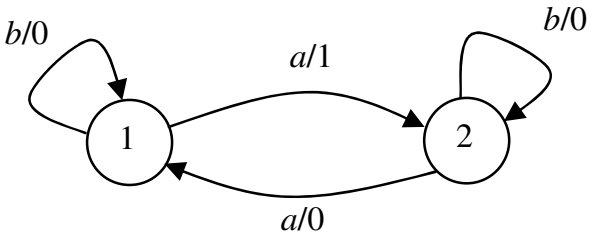


Fig. 1. FSM S_0

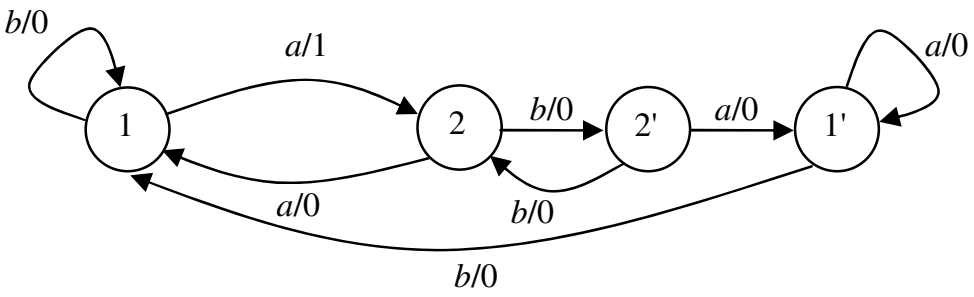


Fig. 2. FSM S_1

domain $\Im_4$, any 4-complete test suite for S_0 should include all input sequences of length three, applied to all states of S_0. In the general case, an m-complete test suite for an FSM with n states should include all input sequences of length $\Delta + 1$, applied to each state. An early work of Moore [11] uses such sequences to establish a lower bound for sequences identifying ‘‘combination lock’’ machines. In fact, the lower bound for the length of an m-complete test suite for an FSM with n states and p inputs is $O(n^3 p^{\Delta+1})$, i.e., it is exponential on the number of extra states [13].

Existing methods, such as W [13] [3], Wp [5], HSI [14] and H [4], which generate an m-complete test suite T for a given minimal deterministic FSM S can be summarized as follows.

Step 1: Determine a minimal initialized state cover K for S.
Step 2: Extend the sequences in K by the (traversal) set $X^{\leq\Delta+1}$.
Step 3: Extend the sequences in $K.X^{\leq\Delta+1}$ in such a way that any two divergent sequences, i.e., reaching distinct states in S, are T-separable.

Existing methods differ mainly in the sequences they use to ensure T-separability in Step 3. In the W method, all sequences in $K.X^{\leq\Delta+1}$ are extended by a characterization set. The Wp method uses a characterization set for sequences in K. $X^{\leq\Delta}$ and state identifiers for the other sequences. The HSI method uses the harmonized state identifiers for all sequences in $K.X^{\leq\Delta+1}$. The H method determines on-the-fly a distinguishing sequence for states reached by each pair of divergent sequences in K. $X^{\leq\Delta+1}$.

We illustrate the generation of a 3-complete test suite for the 2-state FSM in Figure 1 following the strategy used by the existing methods. For this machine, the

characterization set corresponds to a family of harmonized state identifiers $W = H_1 = H_2 = \{a\}$. A minimal state cover for this FSM is $K = \{\varepsilon, a\}$. Then, in the W, Wp, HSI, H methods the sequences in $K.X^{\leq\Delta+1} = pref(\{aaa, aab, aba, abb, ba, bb\})$ are extended by the sequence a. The resulting test suite is $T_1 = pref(\{aaaa, aaba, abaa, abba, baa, bba\})$ of length 28; Figure 3 shows its tree representation, where nodes are labelled with states of the specification FSM and edges are labelled with inputs. Each test corresponds to the sequence of inputs along a path from the root to a node.

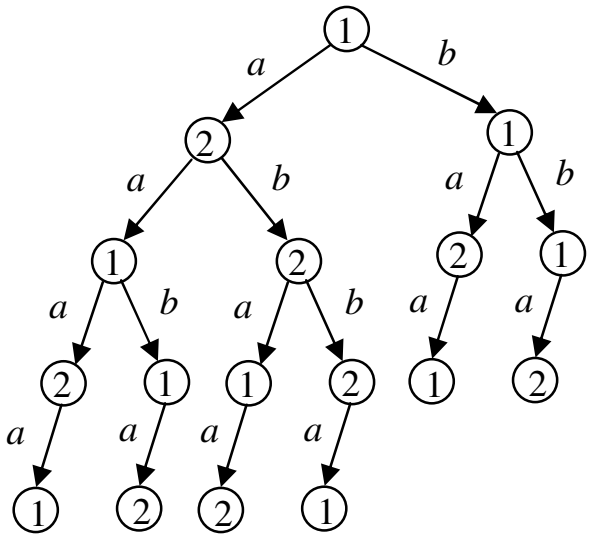


Fig. 3. Tree representation of a 3-complete test suite for S_0

If, in Step 1, shortest transfer sequences are included into a state cover and, in Step 3, shortest distinguishing sequences are used, tests in a resulting m-complete test suite cannot be shortened and if we want to reduce the total length of a test suite we need to find a way of reducing test branching. Indeed, once a test of length l branches into k tests, the test prefix of l inputs contributes kl inputs to the total length of a test suite. For instance, each of tests aa, ab and b branches into two tests in $T1$, thus contributing twice to its total length. In the existing methods, test branching occurs mainly in Step 2, where each test in a minimal state cover is extended by the sequences in the traversal set $X^{\leq\Delta+1}$. As a result of this, such a test branches into at least $|X|^{\Delta+1}$ tests. Apparently, the test length reduction could be achieved by reducing the test branching, which in turn can be performed by distributing the traversal set $X^{\leq\Delta+1}$ over several tests. As soon as one of these tests is a proper prefix of another the overall test branching and thus the test length are reduced. This key observation is illustrated in Figure 4. Assume that test α should be extended by the sequences aa and ba. In Figure 4(a) both sequences extend α, branching the test. Consequently, α contributes twice to the length of the test suite. Suppose that tests α and αb are convergent, and, instead of α, the test αb is extended by aa, as shown in Figure 4(b). We note that this results in a test suite which is, all things being equal, $|\alpha| - 1$ inputs shorter than before. The problem is that an arbitrary distribution of the traversal set may break the m-completeness of a resulting test suite. Thus, we need first to establish conditions for a distribution of the traversal set $X^{\leq\Delta+1}$ such that the m-completeness of a test suite is

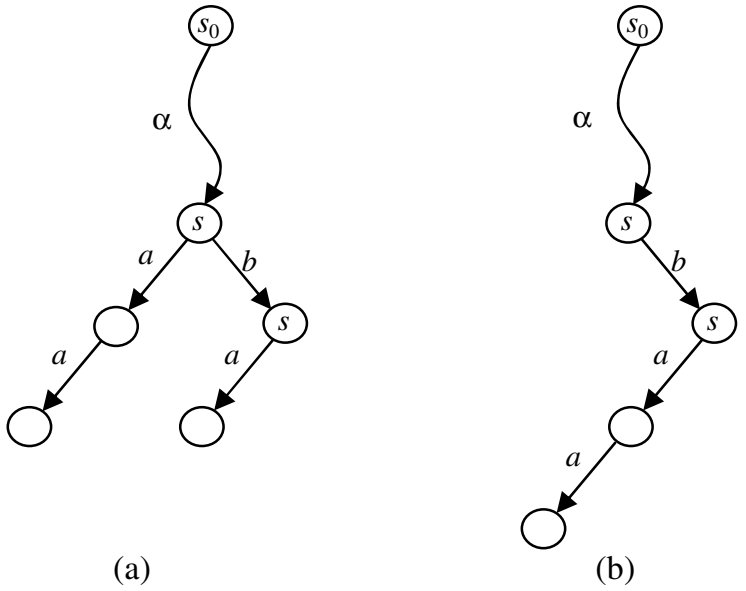


Fig. 4. Test branching (a) *versus* test extension (b)

preserved. The main idea developed in this paper is to distribute it among those tests in a test suite which are convergent, i.e., transfer to the same state, in all FSMs of the fault domain which pass the test suite, reducing test branching.

4 Test Properties

The approach elaborated in this paper is based on properties of FSM tests, namely their convergence and divergence. Recall that two defined input sequences of an FSM converge or diverge if they are transfer sequences for the same state or for different ones, respectively. We generalize these notions to sets of FSMs. Given a non-empty set of FSMs $\Sigma \subseteq \Im$ and two tests $\alpha, \beta \in X^*$, we say that α and β are Σ-*convergent*, if they converge in each FSM of the set Σ. Similarly, we say that α and β are Σ-*divergent*, if they diverge in each FSM of Σ. Two tests are S-convergent (S-divergent) if they are $\{S\}$-convergent ($\{S\}$-divergent). Moreover, when it is clear from the context, we will drop the set in which tests are convergent or divergent.

Test convergence and divergence with respect to a single FSM are complementary, i.e., any two tests are either convergent or divergent. However, when a set of FSMs Σ is considered, some tests are neither Σ-convergent nor Σ-divergent. Notice that the Σ-convergence relation is reflexive, symmetric, and transitive, i.e., it is an equivalence relation over the set of tests. Given a test α, let $[\alpha]$ be the corresponding equivalence class in a non-empty set Σ of FSMs with the same input alphabet. The test convergence and divergence possess the following properties.

Lemma 1. *Given tests α, β, such that $[\alpha] = [\beta]$, the following properties hold:*

(i) $[\alpha\gamma] = [\beta\gamma]$, *for any input sequence* γ.
(ii) *For any test* φ, *if* $[\alpha] \neq [\varphi]$, *then* $[\beta] \neq [\varphi]$.

An important property of T-separable tests is that they are divergent in all FSMs which are T-equivalent to S. Given a test suite T, let $\Im(T)$ be the set of all $Q \in \Im$,

such that $\mathcal{Q}$ and $\mathcal{S}$ are T-equivalent, i.e., $\Im(T)$ is the set of all FSMs in $\Im$ which pass the test suite T.

Lemma 2. *Given a test suite T of an FSM $\mathcal{S}$, T-separable tests are $\Im(T)$-divergent.*
Proof. Let tests α and β be T-separable. Thus, there exist a sequence γ such that $\alpha\gamma$, $\beta\gamma \in T$ and $\lambda_S(\delta_S(s_0, \alpha), \gamma) \neq \lambda_S(\delta_S(s_0, \beta), \gamma)$. Let $\mathcal{Q}$ be an FSM T-equivalent to $\mathcal{S}$; thus, we have that $\lambda_S(\delta_S(s_0, \alpha), \gamma) = \lambda_Q(\delta_Q(q_0, \alpha), \gamma)$ and $\lambda_S(\delta_S(s_0, \beta), \gamma) = \lambda_Q(\delta_Q(q_0, \beta), \gamma)$. It follows that $\lambda_Q(\delta_Q(q_0, \alpha), \gamma) \neq \lambda_Q(\delta_Q(q_0, \beta), \gamma)$. Thus, $\delta_Q(q_0, \alpha) \neq \delta_Q(q_0, \beta)$. ♦

Existing methods for test generation ensure that two tests are divergent by extending them with an appropriate distinguishing sequence. However, Lemmas 1 and 2 indicate that the convergence and divergence of tests also applies to their equivalence classes. It is thus important to identify under which conditions tests are guaranteed to be convergent, i.e., belong to the same equivalence class.

Ensuring convergence is more involved than ensuring divergence; divergence of two tests can be witnessed by different outputs produced in response to a common suffix sequence. The two tests are thus divergent in any FSM T-equivalent to $\mathcal{S}$. However, convergence of two tests cannot be directly ascertained by considering only the two tests. It turns out that the knowledge of the maximum number of states of FSMs in the fault domain allows us to formulate conditions for the convergence of tests. Given a test suite T and a natural number $m \geq n$, let $\Im_m(T) = \Im_m \cap \Im(T)$, i.e., the set of FSMs in $\Im$ which are T-equivalent to $\mathcal{S}$ and have at most m states.

As $\mathcal{S}$ is in the fault domain $\Im_m(T)$, tests which are $\Im_m(T)$-convergent are also $\mathcal{S}$-convergent. Thus, two tests can be $\Im_m(T)$-convergent only if they are $\mathcal{S}$-convergent.

Definition 3. *A set of tests is* $\Im_m(T)$-convergence-preserving *if all its $\mathcal{S}$-convergent tests are $\Im_m(T)$-convergent. Similarly, a set of tests is* $\Im_m(T)$-divergence-preserving *if all its $\mathcal{S}$-divergent tests are $\Im_m(T)$-divergent.*

In other words, a set of tests is $\Im_m(T)$-convergence-preserving if the convergence in the specification FSM is "preserved" in each FSM which passes the test suite T. Similarly, a set of tests is $\Im_m(T)$-divergence-preserving if the divergence in the specification FSM is preserved in each FSM which passes the test suite T.

In the following lemma, the $\Im_m(T)$-convergence relation is considered; thus, $[\alpha]$ is the subset of tests of T which are $\Im_m(T)$-convergent with test α.

Lemma 3. *Given a test suite T for an FSM $\mathcal{S}$ and $\Delta = m - n \geq 0$, let π and φ be $\mathcal{S}$-convergent tests in T, such that, for any sequence υ of length Δ, there exist tests $\alpha \in [\pi]$, $\beta \in [\varphi]$, and an $\Im_m(T)$-divergence-preserving state cover for $\mathcal{S}$ in T containing $\{\alpha, \beta\}.pref(\upsilon)$. Then, π and φ are $\Im_m(T)$-convergent.*

Proof. Suppose that π and φ are not $\Im_m(T)$-convergent. Thus, there exists $\mathcal{Q} \in \Im_m(T)$, such that π and φ are $\mathcal{Q}$-divergent. As π and φ are $\mathcal{S}$-convergent, the FSM $\mathcal{Q}$ is not equivalent to $\mathcal{S}$ and there must exist an input sequence γ such that $\mathcal{S}$ and $\mathcal{Q}$ are $\{\pi\gamma, \varphi\gamma\}$-distinguishable. Assume that γ is a shortest input sequence with this property. Thus,

$$\mathcal{S} \text{ and } \mathcal{Q} \text{ are } (([\pi] \cup [\varphi]).\gamma')\text{-equivalent, for all } \gamma', \text{ such that } |\gamma'| < |\gamma|. \qquad (1)$$

We have that $|\gamma| > \Delta$, since otherwise there would exist $\alpha' \in [\pi]$ and $\beta' \in [\varphi]$ such that $\{\alpha'\gamma, \beta'\gamma\} \subseteq T$, implying that $\mathcal{S}$ and $\mathcal{Q}$ are T-distinguishable.

Let $\alpha \in [\pi]$ and $\beta \in [\varphi]$ be such that there exists an $\mathfrak{I}_m(T)$-divergence-preserving state cover for $\mathcal{S}$ in T containing the set $\{\alpha, \beta\}.pref(\gamma_\Delta)$, where γ_i is the prefix of γ of length i. Without loss of generality, we assume that $\mathcal{S}$ and $\mathcal{Q}$ are $\{\alpha\gamma\}$-distinguishable, i.e., $\lambda_{\mathcal{Q}}(q_0, \alpha\gamma) \neq \lambda_{\mathcal{S}}(s_0, \alpha\gamma)$. Let $A_i = \{\alpha, \beta\}.pref(\gamma_i)$, $0 \leq i \leq \Delta$. The tests $\alpha\gamma_i$ and $\beta\gamma_i$ are $\mathcal{Q}$-divergent and, moreover, A_i is $\mathfrak{I}_m(T)$-divergence-preserving. We show by induction that, for all $0 \leq i \leq \Delta$, $|\delta_{\mathcal{Q}}(q_0, A_i)| \geq i + |\delta_{\mathcal{S}}(s_0, A_i)| + 1$.

Base case: For $i = 0$, we have that $A_0 = \{\alpha, \beta\}$. As α and β are $\mathcal{S}$-convergent and $\mathcal{Q}$-divergent, the result follows, since $|\delta_{\mathcal{Q}}(q_0, A_0)| = 2$ and $|\delta_{\mathcal{S}}(s_0, A_0)| = 1$.

Inductive Step: Suppose that the result holds i, $0 \leq i < \Delta$, i.e.,

$$|\delta_{\mathcal{Q}}(q_0, A_i)| \geq i + |\delta_{\mathcal{S}}(s_0, A_i)| + 1. \tag{2}$$

We show that the result holds for $i + 1$. Let $j \leq i$. Suppose that $\alpha\gamma_{i+1}$ and $\alpha\gamma_j$ are $\mathcal{S}$-divergent; then $\alpha\gamma_{i+1}$ is $\mathcal{Q}$-divergent with $\alpha\gamma_j$ and $\beta\gamma_j$, since A_{i+1} is $\mathfrak{I}_m(T)$-divergence-preserving. Suppose now that $\alpha\gamma_{i+1}$ and $\alpha\gamma_j$ are $\mathcal{S}$-convergent. Let χ be the suffix which extends γ_{i+1} in γ, i.e., $\gamma = \gamma_{i+1}\chi$. If $\alpha\gamma_{i+1}$ is $\mathcal{Q}$-convergent with $\alpha\gamma_j$, then $\alpha\gamma_j\chi$ distinguishes $\mathcal{S}$ and $\mathcal{Q}$, since $\lambda_{\mathcal{Q}}(q_0, \alpha\gamma_j\chi) = \lambda_{\mathcal{Q}}(q_0, \alpha\gamma_{i+1}\chi) = \lambda_{\mathcal{Q}}(q_0, \alpha\gamma) \neq \lambda_{\mathcal{S}}(s_0, \alpha\gamma) = \lambda_{\mathcal{S}}(s_0, \alpha\gamma_{i+1}\chi) = \lambda_{\mathcal{S}}(s_0, \alpha\gamma_j\chi)$. As $|\gamma_j\chi| < |\gamma_{i+1}\chi| = |\gamma|$, it follows that $\alpha\gamma_{i+1}$ should be $\mathcal{Q}$-divergent with $\alpha\gamma_j$ and $\beta\gamma_j$, since otherwise we have a contradiction to (1). By the same token, the test $\alpha\gamma_{i+1}$ is $\mathcal{Q}$-divergent with $\beta\gamma_j$. Thus, $\alpha\gamma_{i+1}$ is $\mathcal{Q}$-divergent with $\alpha\gamma_j$, $j \leq i$, i.e., with all tests in A_i and reaches a state in $\mathcal{Q}$ which is not reached by the tests in A_i. Hence,

$$|\delta_{\mathcal{Q}}(q_0, A_{i+1})| \geq |\delta_{\mathcal{Q}}(q_0, A_i)| + |\delta_{\mathcal{Q}}(q_0, \alpha\gamma_{i+1})| \geq |\delta_{\mathcal{Q}}(q_0, A_i)| + 1. \tag{3}$$

If $\alpha\gamma_{i+1}$ is $\mathcal{S}$-convergent with some test in A_i, then

$$|\delta_{\mathcal{S}}(s_0, A_{i+1})| = |\delta_{\mathcal{S}}(s_0, A_i)|. \tag{4}$$

The induction thus applies, since

$$\begin{aligned}
&|\delta_{\mathcal{Q}}(q_0, A_i)| \geq i + |\delta_{\mathcal{S}}(s_0, A_i)| + 1 && \text{(inductive hypothesis (2))}\\
&|\delta_{\mathcal{Q}}(q_0, A_i)| + 1 \geq (i + 1) + |\delta_{\mathcal{S}}(s_0, A_i)| + 1 &&\\
&|\delta_{\mathcal{Q}}(q_0, A_{i+1})| \geq (i + 1) + |\delta_{\mathcal{S}}(s_0, A_{i+1})| + 1 && \text{(due to (3) and (4))}
\end{aligned}$$

On the other hand, if $\alpha\gamma_{i+1}$ is $\mathcal{S}$-divergent with all tests in A_i, then

$$|\delta_{\mathcal{S}}(s_0, A_{i+1})| = |\delta_{\mathcal{S}}(s_0, A_i)| + 1 \tag{5}$$

In this case, $\beta\gamma_{i+1}$ is also $\mathcal{Q}$-divergent with all tests in A_i, since A_{i+1} is $\mathfrak{I}_m(T)$-divergence-preserving. Moreover, $\beta\gamma_{i+1}$ is $\mathcal{Q}$-divergent with $\alpha\gamma_{i+1}$. Thus, we have that

$$|\delta_{\mathcal{Q}}(q_0, A_{i+1})| = |\delta_{\mathcal{Q}}(q_0, A_i)| + |\delta_{\mathcal{Q}}(q_0, \{\alpha\gamma_j, \beta\gamma_j\})| \geq |\delta_{\mathcal{Q}}(q_0, A_i)| + 2 \tag{6}$$

The induction also applies, since

$$\begin{aligned}
&|\delta_{\mathcal{Q}}(q_0, A_i)| \geq i + |\delta_{\mathcal{S}}(s_0, A_i)| + 1 && \text{(inductive hypothesis (2))}\\
&|\delta_{\mathcal{Q}}(q_0, A_i)| + 2 \geq (i + 1) + (|\delta_{\mathcal{S}}(s_0, A_i)| + 1) + 1 &&\\
&|\delta_{\mathcal{Q}}(q_0, A_{i+1})| \geq (i + 1) + |\delta_{\mathcal{S}}(s_0, A_{i+1})| + 1 && \text{(due to (5) and (6))}
\end{aligned}$$

This concludes the induction proof. Then, for all $0 \le i \le \Delta$, it holds that $|\delta_Q(q_0, A_i)| \ge i + |\delta_S(s_0, A_i)| + 1$. In particular, the set of tests A_Δ reaches at least $\Delta + |\delta_S(s_0, A_\Delta)| + 1$ states in $\mathcal{Q}$.

Consider now a smallest set K, such that $K \cup A_\Delta$ is an $\mathfrak{I}_m(T)$-divergence-preserving state cover for $\mathcal{S}$ in T; thus, $|K| = n - |\delta_S(s_0, A_\Delta)|$, since α and β are $\mathcal{S}$-convergent. As $K \cup A_\Delta$ is $\mathfrak{I}_m(T)$-divergence-preserving, the tests of the set K reach exactly $n - |\delta_S(s_0, A_\Delta)|$ states in $\mathcal{Q}$, and each of them is distinct from all states reached by A_Δ. Thus, the tests in $K \cup A_\Delta$ reach at least $n - |\delta_S(s_0, A_\Delta)| + \Delta + |\delta_S(s_0, A_\Delta)| + 1 = n + m - n + 1 = m + 1$ states in $\mathcal{Q}$, contradicting the fact that $\mathcal{Q}$ has at most m states. ♦

The importance of Lemma 3 for test generation is that it shows how to ensure the $\mathfrak{I}_m(T)$-convergence of two $\mathcal{S}$-convergent tests. This in turn, allows including these tests into the same equivalence class. Then, Lemma 1 can be applied, which indicates that if a test should be extended by given sequences, e.g., from the traversal set, any tests of its equivalence class can be chosen, distributing these sequences over several tests. Lemma 3 also leads to the necessary and sufficient conditions for test completeness with respect to the fault domain $\mathfrak{I}_m$, where each FSM has at most m states, $m \ge n$.

Theorem 1. *Let T be a test suite for an FSM $\mathcal{S}$ with n states and $m \ge n$. Then, the following statements are equivalent:*

(i) T is an m-complete test suite for $\mathcal{S}$

(ii) T contains an $\mathfrak{I}_m(T)$-convergence-preserving initialized transition cover for $\mathcal{S}$.

Proof

(ii) $\Rightarrow$ (i) Let T contain an $\mathfrak{I}_m(T)$-convergence-preserving initialized transition cover A for $\mathcal{S}$, and $\mathcal{Q} \in \mathfrak{I}_m(T)$. Define the relation $h \subseteq S \times Q$ as follows:

$$(s, q) \in h \Leftrightarrow \text{there exists } \alpha \in A, \text{ such that } \delta_S(s_0, \alpha) = s \text{ and } \delta_Q(q_0, \alpha) = q.$$

As A is a transition cover for $\mathcal{S}$, for each $s \in S$ there exists $q \in Q$ such that $(s, q) \in h$. Moreover, as A is $\mathfrak{I}_m(T)$-convergence-preserving, for each $s \in S$, there exists only one $q \in Q$ such that $(s, q) \in h$; thus, h is a mapping. As $\varepsilon \in A$,

$$h(s_0) = q_0.$$

Let $s \in S$ and $x \in X$. As A is a transition cover for $\mathcal{S}$,

$$\text{there exists } \alpha x \in A \text{ such that } \delta_S(s_0, \alpha) = s.$$

Correspondingly,

$$h(\delta_S(s_0, \alpha), x) = h(\delta_S(s_0, \alpha x)) = \delta_Q(q_0, \alpha x) = \delta_Q(\delta_Q(q_0, \alpha), x) = \delta_Q(h(\delta_S(s_0, \alpha)), x)$$

and

$$\lambda_S(\delta_S(s_0, \alpha), x) = \lambda_Q(\delta_Q(q_0, \alpha), x) = \lambda_Q(h(\delta_S(s_0, \alpha)), x),$$

as $\mathcal{Q} \in \mathfrak{I}_m(T)$.

Thus, h is an isomorphism and, as $h(s_0) = q_0$, it follows that $\mathcal{Q}$ is equivalent to $\mathcal{S}$.

(i) $\Rightarrow$ (ii) Let T be an m-complete test suite. First, notice that any m-complete test suite is a transition cover for the FSM $\mathcal{S}$. Otherwise, there exists a transition of $\mathcal{S}$ which is not traversed by the test suite; an FSM that is T-equivalent to, but

distinguishable from, $\mathcal{S}$ can be obtained from $\mathcal{S}$ by mutating the output in this transition. By definition, T is prefix closed, thus, it is an initialized transition cover.

As T is an m-complete test suite, each FSM $\mathcal{Q} \in \Im_m(T)$ is equivalent to $\mathcal{S}$, i.e., there exists a mapping h: $S \rightarrow Q$ such that $h(s_0) = q_0$ and for each transition (s, x) it holds that

$$h(\delta_S(s, x)) = \delta_Q(h(q), x)$$

and thus, since $h(s_0) = q_0$, for each input sequence α it holds that

$$h(\delta_S(s_0, \alpha)) = \delta_Q(h(s_0), \alpha) = \delta_Q(q_0, \alpha).$$

Let α and β be $\mathcal{S}$-convergent, i.e., $\delta_S(s_0, \alpha) = \delta_S(s_0, \beta)$. It follows that

$$\delta_Q(q_0, \alpha) = h(\delta_S(s_0, \alpha)) = h(\delta_S(s_0, \beta)) = \delta_Q(q_0, \beta).$$

Thus, α and β are also $\mathcal{Q}$-convergent and, consequently, the set is $\Im_m(T)$-convergence-preserving. ♦

Considering the generation methods discussed in Section 3, we note that the conditions of Lemma 3 are satisfied for all pairs of $\mathcal{S}$-convergent tests in $K \cup K.X$, which turns out to be a transition cover for $\mathcal{S}$. Thus, the test suites generated by these methods satisfy the conditions of Theorem 1, since $K \cup K.X$ is an initialized transition cover. At the same time, Theorem 1 suggests that rather than considering the whole set of tests in $K.X^{\leq \Delta+1}$ at once, as the existing methods do, it is sufficient to ensure convergence of tests covering all transitions, using Lemma 3. Moreover, Lemmas 1, 2, and 3 indicate that this can be achieved in an iterative way, namely, the convergence for tests covering a current transition can be ensured based on the convergence established for other transitions. In the next section we elaborate this idea in a method for complete test suite generation.

5 Test Generation Method

In this section, we present a method, called SPY-method, which generates an m-complete test suite by building an $\Im_m(T)$-convergence-preserving transition cover. In the method, the knowledge about test convergence and divergence obtained during the execution helps identify the possibility of extending tests already in the test suite. Such an extension avoids branching of tests and thus contributes to test suite shortening. During the execution of the method, the $\Im_m(T)$-convergence of tests is determined. Notice that any two $\Im_m(T)$-convergent tests are also $\Im_m(T')$-convergent, for each $T' \supseteq T$. Thus, the inclusion of new tests in T does not invalidate this property.

As the $\Im_m(T)$-convergence relation is an equivalence relation, it can be represented by the partition it induces. In a given stage of the method execution only a subset of the $\Im_m(T)$-convergence relation might be known. We denote by Π the partition induced by the pairs of tests which are known to be $\Im_m(T)$-convergent. Given a test $\alpha \in T$, we denote by $[\alpha]_\Pi$ the block of the partition Π which contains α.

We assume that a family H of harmonized state identifiers is provided. Given a test α, let $H(\alpha) \in H$ be the state identifier for state $\delta_S(s_0, \alpha)$. The method starts by determining a minimal initialized state cover K, as in Step 1 of existing methods.

Then, the tests in K are extended by the appropriate state identifiers. Each block in the initial partition Π is a singleton, since no convergence is initially known. The method iterates until the set of tests which are $\Im_m(T)$-convergent with the tests in K becomes a transition cover for S.

During the execution of the method, it is necessary to extend two tests in T to ensure their divergence. As the divergence of tests also applies to other tests in their blocks, when more than one test is available in a given block, the one which will result in a shorter test suite is selected. This is achieved as follows. Suppose that test $\alpha \notin T$ should be added to T. Let β be the longest prefix of α which is in T. If β is not a proper prefix of another test in T, we have that $len(T \cup \{\alpha\}) = len(T) + |\alpha| - |\beta|$, i.e., adding α to T results only in extending the test β by $|\alpha| - |\beta|$ input symbols. On the other hand, if β is a proper prefix of some other test in T, it holds that $len(T \cup \{\alpha\}) = len(T) + |\alpha| + 1$, as it results in an additional testing branching. Thus, selection of a test which has to be extended by some input sequence, e.g., a state distinguishing sequence, should result, whenever possible, in extending some test in T that is not a proper prefix of another test.

After adding new tests two blocks containing tests that are $\Im_m(T)$-convergent, are merged, i.e., replaced by their union, iteratively. The merge of blocks can result in a new partition for which the $\Im_m(T)$-convergence of other tests can be concluded, due to the application of Lemma 1(i) and thus, the procedure of merging should be repeated. We denote by $closure(\Pi)$ the partition obtained after merging the blocks of Π as much as possible, by applying subset merging and Lemma 1(i).

We now present SPY-method.

Input: An FSM S with n states, a family of harmonized state identifiers H and a natural number $m \geq n$.
Output: An m-complete test suite.
Determine a minimal initialized state cover K.
$T := pref(\{\alpha.H(\alpha) \mid \alpha \in K\})$
$\Pi := \{\{\alpha\} \mid \alpha \in T\}$
While there exists a transition (s, x) not covered by the set of tests in T which are $\Im_m(T)$-convergent with some test in K
    Let $\alpha, \beta \in K$ be such that $\delta_S(s_0, \alpha) = s$ and $\delta_S(s_0, \beta) = \delta_S(s, x)$
    For each $\gamma \in X^{\leq \Delta}$, each $\sigma \in H(\beta\gamma)$
        Select $\alpha' \in [\alpha]_\Pi$, such that $len(T \cup \{\alpha' x\gamma\sigma\})$ is minimal
        $T := T \cup pref(\alpha' x\gamma\sigma)$
        Select $\beta' \in [\beta]_\Pi$, such that $len(T \cup \{\beta'\gamma\sigma\})$ is minimal
        $T := T \cup pref(\beta'\gamma\sigma)$
        $\Pi := closure(\Pi \cup \{\{\chi\} \mid \chi \in \{\alpha' x, \beta'\}.pref(\gamma\sigma)\})$
    End for
    $\Pi := closure(\Pi \cup \{[\alpha x]_\Pi \cup [\beta]_\Pi\})$
End while
Return T

Theorem 2. *SPY-method generates an m-complete test suite T for S.*

Proof. Let $C = \{\beta \in [\alpha]_\Pi \mid \alpha \in K\}$, i.e., C is the set of tests which are $\Im_m(T)$-convergent with some test in K. Notice that C is $\Im_m(T)$-convergence-preserving, since by its definition, any two tests in C which are S-convergent are also $\Im_m(T)$-convergent. We first show that in each iteration of the method, C is extended to cover a transition (s, x) which was not yet covered. Let $\alpha, \beta \in K$ be such that $\delta_S(s_0, \alpha) = s$ and $\delta_S(s_0, \beta) = \delta_S(s, x)$. The method then uses the state identifiers required to ensure that tests αx and β are $\Im_m(T)$-convergent. Indeed, for all $\gamma \in X^{\leq\Delta}$, tests α' and β', which are $\Im_m(T)$-convergent with α and β, respectively, are selected and the tests $\alpha' x\gamma$ and $\beta'\gamma$ are extended with the corresponding state identifiers. As the state cover K is also extended by the state identifiers, we have that for each sequence γ of length Δ, the set $K \cup \{\alpha x, \beta\}.pref(\gamma)$ is $\Im_m(T)$-divergence-preserving; thus, the conditions of Lemma 3 are satisfied and tests αx and β are $\Im_m(T)$-convergent. The the blocks containing αx and β are merged. As a result, the transition (s, x) is covered by C. When the method terminates, C is a transition cover for S. As K is initialized and $K \subseteq C$, C is also initialized. Hence, by Theorem 1, T is m-complete, since $C \subseteq T$ is an $\Im_m(T)$-convergence-preserving initialized transition cover. ♦

In each iteration the proposed method deals with the set $X^{\leq\Delta}$, while the theoretical results indicate that an m-complete test suite should include all sequences in the traversal set $X^{\leq\Delta+1}$ [11] [8]. Notice, however, that to obtain a transition cover as required by Theorem 1, the tests of a state cover has to be extended by X, which is in its turn extended by $X^{\leq\Delta}$. Therefore, all sequences in the traversal set $X^{\leq\Delta+1}$ are indeed present in the resulting test suite. Nevertheless, the distribution of the traversal set over several tests usually results in shorter test suites, as demonstrated by the example and the experimental results on the next sections.

Compared with the existing methods for m-complete test suite generation, SPY-method requires the additional operations of handling the partitions of tests and selecting the tests in a partition which lead to a minimal length increase. We discuss the overhead imposed by these operations. The partitions used in the method can be efficiently handled using a *union-find* structure [6]. The operation of merging blocks and determining to which block a test belongs can be performed in $O(Ack^{-1}(l, l))$, where $Ack^{-1}(l, l)$ is the inverse of the extremely quickly-growing Ackermann function [6]. For any reasonable value of l, $Ack^{-1}(l, l)$ is less than five, i.e., the running time of these operations is effectively a small constant. In order to efficiently calculate a length increase caused by new tests, test suites can be represented by trees. Then it is possible to identify when a test will create a new test (branching at a non-leaf node) or extend an existing one (extending a leaf node) by retrieving the information about nodes in the tree. As the size of the tree is proportional to the length of the test suite, the overhead imposed by the additional operations required by the method, i.e., maintaining the partitions and determining the length increase, is polynomial in the length of the test suite.

6 Example

In this section, we illustrate the execution of the method. Consider the FSM in Figure 1. We generate a 3-complete test suite, using the family of harmonized state identifiers as in Section 3, $H_1 = H_2 = \{a\}$. Note that as before, $n = 2$ and $m = 3$.

The method determines a minimal initialized state cover $K = \{\varepsilon, a\}$. The test suite is initialized with $T := \{\alpha.H(\alpha) \mid \alpha \in K\} = pref(aa)$ and the partition $\Pi := \{\{\varepsilon\}, \{a\}, \{aa\}\}$. Notice that the tests in K already cover the transition $(1, a)$. Then, the method iterates until all the other transitions are also covered by the tests which are $\Im_m(T)$-convergent with either ε or a. Notice that in this example, both H_1 and H_2 contain only the sequence a. Therefore, each state identifier used in the method is always equal to $\{a\}$, i.e., $\sigma = a$ throughout this example.

The method selects the transition $(s, x) = (1, b)$; thus $\alpha = \beta = \varepsilon$. At this stage each block is a singleton; thus, selecting the empty sequence ε is the only option in the first iteration. For each $\gamma \in X^{\leq\Delta} = \{\varepsilon, a, b\}$, the test ε is extended by $x\gamma\sigma$ and $\gamma\sigma$; namely, the empty sequence is extended by the sequences ba, a, baa, aa, bba, and ba. The test suite becomes $T = pref(\{aa, baa, bba\})$ and the partition Π is updated to include the new tests (each of them also becomes a singleton block in the partition). According to Lemma 3, ε and b are now $\Im_m(T)$-convergent, thus, blocks $\{\varepsilon\}$ and $\{b\}$ should be merged. After updating the partition and determining its closure, the partition $\Pi = \{\{\varepsilon, b, bb\}, \{a, ba, bba\}, \{aa, baa\}\}$ is obtained. The resulting test suite is represented in Figure 4. The nodes with the same color are in the same block of the partition Π.

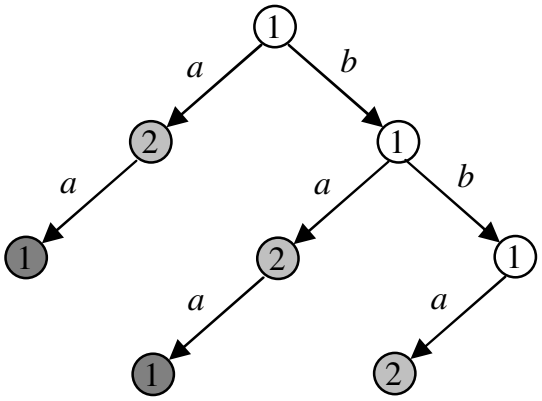


Fig. 4. Tree representation of $pref(\{aa, baa, bba\})$

The methods selects the transition $(s, x) = (2, a)$. Then $\alpha = a$ and $\beta = \varepsilon$. In this iteration, the blocks of the partition contain several tests; thus, there are choices when selecting the test which is extended by the state identifier. For each $\gamma \in X^{\leq\Delta} = \{\varepsilon, a, b\}$, some test in $[\alpha]_\Pi = [a]_\Pi$ should be extended by $x\gamma\sigma$ and some test in $[\beta]_\Pi = [\varepsilon]_\Pi$ should be extended by $\gamma\sigma$. For $\gamma = \varepsilon$ some test in $[a]_\Pi = \{a, ba, bba\}$ has to be extended by $x\gamma\sigma = aa$; the test suites resulting from extending a, ba and bba by aa have lengths 12, 12 and 13, respectively. Thus, the test a is selected and aaa is added to T. Then, some test in $[\varepsilon]_\Pi = \{\varepsilon, b, bb\}$ should be extended by a. As $a \in T$, no additional test is included. For $\gamma = a$, some test in $[a]_\Pi$ has to be extended by $x\gamma\sigma = aaa$ and some test in $[\varepsilon]_\Pi$ by $\gamma\sigma = aa$. A test suite of shorter length can be obtained by extending either a or ba. The test a is selected and $aaaa$ is added to T. There is no need to extend any sequence in $[\varepsilon]_\Pi$ by $\gamma\sigma = aa$, since $aa \in T$. For $\gamma = b$, $\sigma = a$, some test in $[a]_\Pi$ should be extended by $x\gamma\sigma = aba$ and some test in $[\varepsilon]_\Pi$ by $\gamma\sigma = ba$.

Extending tests *a*, *ba* and *bba* by *aba* results in test suites of lengths 17, 15 and 16, respectively. The test *ba* is, then, selected and *baaba* is added to *T*. Again, there is no need to extend any sequence in $[\varepsilon]_\Pi$ by $\gamma\sigma = ba$. The test suite becomes $T = pref(\{aaaa, baaba, bba\})$. The tests ε and *aa* are now $\Im_m(T)$-convergent and thus, blocks {ε, *b*, *bb*} and {*aa*, *baa*} should be merged. After merging these blocks and deriving the closure, the partition $\Pi = \{\{\varepsilon, aa, aaaa, b, baa, baab, bb\}, \{a, aaa, ba, baaba, bba\}\}$ is obtained. Figure 5 represents the resulting test suite.

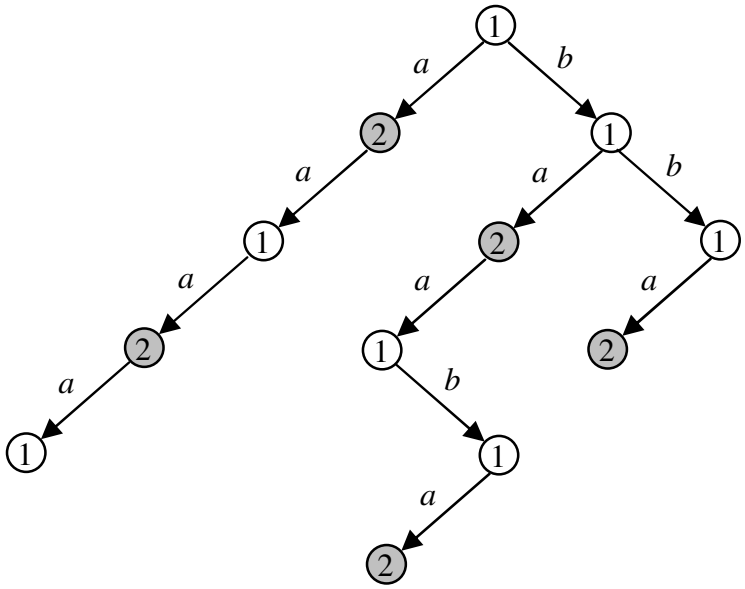


Fig. 5. Tree representation of $pref(\{aaaa, baaba, bba\})$

It remains to cover the transition $(s, x) = (2, b)$; thus $\alpha = \beta = a$. For $\gamma = \varepsilon$, some test in $[a]_\Pi = \{a, aaa, ba, baaba, bba\}$ should be extended by $x\gamma\sigma = ba$ and $\gamma\sigma = a$. The test suites obtained by extending either test *baaba* or *bba* by *ba* have the same length; the test *bba* is then selected and *bbaba* is added to *T*. Some test in $[a]_\Pi$ has to be extended by $\gamma\sigma = a$, which does not need any additional test, since $aa \in T$. For $\gamma = a$, some test in $[a]_\Pi$ should be extended by $x\gamma\sigma = baa$ and $\gamma\sigma = aa$. The test suite of a shorter length is obtained by extending *bba* by *baa* and the test *bbabaa* is added to *T*. There is no need to extend any test in $[a]_\Pi$ by *aa*, since $aaa \in T$. For $\gamma = b$, some test in $[a]_\Pi$ should be extended by $x\gamma\sigma = bba$ and $\gamma\sigma = ba$. The test suite of a shorter length is obtained by extending *baaba* by *bba* and the test *baababba* is added to *T*. To extend some test in $[a]_\Pi$ by $\gamma\sigma = ba$, no additional test is required, since $baaba \in T$ and $baa \in [a]_\Pi$. The resulting test suite is $T = pref(\{aaaa, baababba, bbabaa\})$ of length 21. Recall that the test suite T_1 obtained by the existing methods for the machine in Figure 1 has length 28.

7 Experimental Results

In this section, we present the results of an experiment with the HSI method and the proposed method, comparing the length of the test suites they generate. We randomly generate minimal FSMs with five inputs, five outputs and the number of states *n* ranging from five to 50. We executed both the HSI method and the proposed method

for generating m-complete test suites, for $n \leq m \leq n + 3$ and calculated the ratio of reduction, i.e., the average ratio of the length of the test suite generated by SPY-method and the length of the test suite generated by the HSI method. For each setting (values of n and m), we generated 30 FSMs and the respective test suites, totalling 5520 FSMs. In Figure 6, we plot the variation of the average ratio with respect to the number of states. We notice that the test suites generated by our method are on average up to 40% shorter than the test suites obtained by the HSI method; moreover, the larger the number of states in the specification FSM and the number of extra states in implementations, the bigger the reduction.

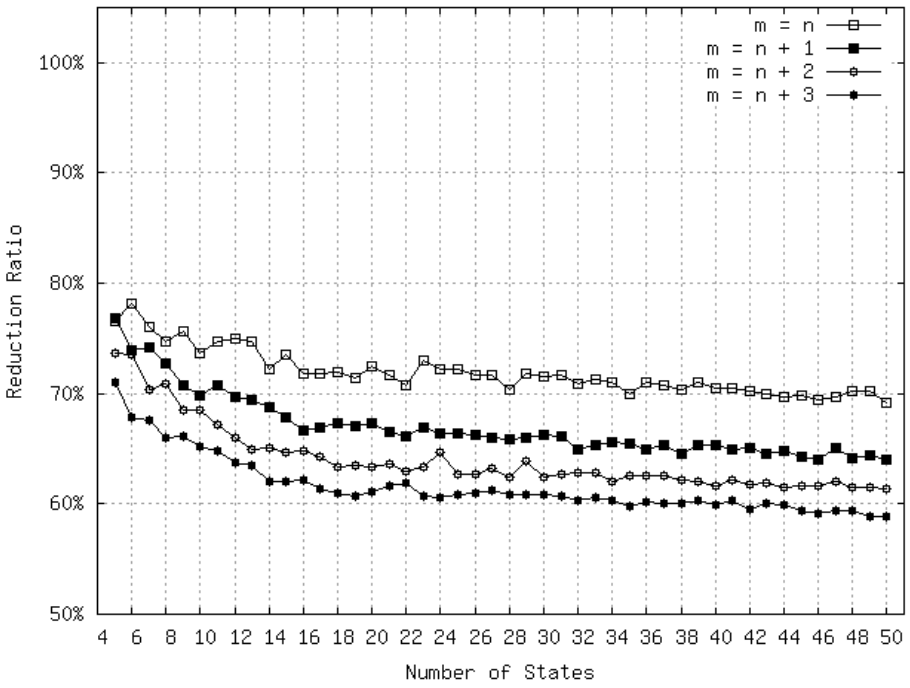


Fig. 6. Average reduction ratio

8 Conclusions

In this paper we investigated the problem of generating m-complete test suites for an FSM with n states, when implementation FSMs may have extra states.

The main contributions of this paper are as follows. Firstly, although we have not refuted the inevitability of including the sequences of a traversal set in an m-complete test suite, we showed that these sequences can be arranged in such a way that test branching is significantly reduced. Secondly, we stated conditions which guarantee that the resulting test suite is indeed m-complete and elaborated a test generation method based on these conditions. Differently from all existing methods, the proposed method distributes the sequences in the traversal set over several tests avoiding as much as possible test branching and thus leading to shortening of the resulting test suite. Finally, we experimentally compared the proposed method with the HSI-method. The experimental results indicate that obtained tests are on average up to 40% shorter.

As future work, it is possible to combine the on-the-fly determination of distinguishing sequences used in the H method with the possibility of distributing them. Another possible extension is the further investigation of properties of test convergence and divergence.

Acknowledgements

The authors acknowledge financial supports of NSERC (Grant OGP0194381), Brazilian Funding Agency CNPq (Grant 200032/2008-9), and FCP Russian program (contract 02.514.12.4002).

References

1. Bochmann, G.v., Petrenko, A.: Protocol testing: review of methods and relevance for software testing. In: ACM International Symposium on Software Testing and Analysis (ISSTA 1994), pp. 109–124. ACM Press, New York (1994)
2. Binder, R.: Testing Object-Oriented Systems. Addison-Wesley, Inc., Reading (2000)
3. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. 4(3), 178–187 (1978)
4. Dorofeeva, R., El-Fakih, K., Yevtushenko, N.: An improved conformance testing method. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 204–218. Springer, Heidelberg (2005)
5. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Trans. Softw. Eng. 17(6), 591–603 (1991)
6. Galil, Z., Italiano, G.F.: Data structures and algorithms for disjoint set union problems. ACM Comput. Surv. 23(3), 319–344 (1991)
7. Gonenc, G.: A method for the design of fault detection experiments. IEEE Trans. on Comput. 19(6), 551–558 (1970)
8. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - a survey. Proceedings of the IEEE. 84(8), 1090–1123 (1996)
9. Hennie, F.C.: Fault-detecting experiments for sequential circuits. In: Proceedings of Fifth Annual Symposium on Circuit Theory and Logical Design, Princeton, New Jersey, pp. 95–110 (1965)
10. Hierons, R.M., Ural, H.: Optimizing the length of checking sequences. IEEE Trans. on Comput. 55(5), 618–629 (2006)
11. Moore, E.F.: Gedanken-experiments on sequential machines. Automata Studies, Annals of Mathematical Studies (34), 129–153 (1956)
12. Petrenko, A., Higashino, T., Kaji, T.: Handling redundant and additional states in protocol testing. In: IFIP 8th Inter. Workshop on Protocol Test Systems, pp. 307–322. Chapman & Hall, Boca Raton (1995)
13. Vasilevskii, M.P.: Failure diagnosis of automata. Cybernetics 4, 653–665 (1973)
14. Yevtushenko, N., Petrenko, A.: Synthesis of test experiments in some classes of automata. Automatic Control and Computer Sciences 24(4), 50–55 (1990)

Apêndice J

A. L. Bonifácio, A. Moura, A. S. Simão. A Generalized Model-based Test Generation Method. In: The 6th IEEE International Conferences on Software Engineering and Formal Methods, p. 139-148, Cape Town, Africa do Sul, 2008.

2008 Sixth IEEE International Conference on Software Engineering and Formal Methods

A Generalized Model-based Test Generation Method

Adilson Luiz Bonifácio* Arnaldo Vieira Moura†
Computing Institute, University of Campinas
P.O. 6176 – Campinas – Brazil – 13081-970
adilson@ic.unicamp.br arnaldo@ic.unicamp.br

Adenilso da Silva Simão
Mathematic Science and Computing Institute, University of São Paulo
P.O. 668 – São Carlos – Brazil – 13560-970
adenilso@icmc.usp.br

Abstract

In this paper we present a generalization to the W-method [3], which can be used for automatically generating test cases. In contrast to the W-method, this generalization allows for test case generation even in the absence of characterization sets for the specification. We give proofs of correctness for this generalization, and show how to derive the original W-method from it as a particular case. Proofs of correctness for the W-method, not given in the original paper, are also presented in a clear and detailed way.

1 Introduction

Conformance testing aims at demonstrating that the implementation behavior conforms to the behavior dictated by the specification [7, 20, 21]. In the literature, there are many model-based test derivation methods for conformance testing of critical and reactive systems [4, 14, 22]. The problem of generating test cases for conformance testing has been intensively studied, specially for models based on Finite State Machines (FSMs) [5, 9, 10, 11, 19, 23]. One of the most well-known of these test generation methods is the W-method [3], which uses the notion of characterization sets. The W-method was proposed for deterministic FSMs and it has been widely investigated, and many variations have been developed around its main ideas [11, 12, 13, 18].

In this paper we present a generalization of the W-method. This generalization allows us to derive a m-complete test suite without using a characterization set. A test suite is m-complete if it guarantees a complete fault coverage[17], while considering deterministic FSM implementations with up to m states. In fact, our method can generate test suites using only subsets of any characterization set. We discuss how to refine the generalization in order to arrive at the original W-method, demonstrating that the latter is a particular case of our method. Proofs of correctness are presented in a clear form, including the correctness for the original W-method.

This paper is organized as follows. In Section 2 we describe some work related to our proposal, and we review some basic concepts. The concept of transition covers for FSMs is presented in Section 3. In Section 4 we introduce equivalence in FSMs, and stratified families of sets. The generation of a complete test suite is presented in Section 5. In Section 6 we reconsider characterization sets. How to refine our method in order to obtain the original W-method is described in Section 7. In Section 8 we present the algorithm for the generalized method, and illustrate its usefulness with an example in Section 9. Finally, in Section 10, we give some concluding remarks.

2 Related Works

This section reviews the FSM model and some important related notions. We also present more details about the W-method and other variant model-based test generation methods, such as the W_p and HSI methods.

2.1 Finite State Machines

The basic model used to capture a system behavior is the FSM. Formally, a FSM [8] is a system $M = (X, Y, S, s_0, \delta, \lambda)$ given by:

- a finite input alphabet, X;

*Supported by CNPq grant 141978/2008-2
†Supported by CNPq grant 472504/2007-0

978-0-7695-3437-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SEFM.2008.17

- a finite output alphabet, Y;
- a finite set of states, S;
- an initial state $s_0 \in S$; and
- output and transition functions, respectively, $\lambda : X \times S \to Y$ and $\delta : X \times S \to S$.

Note that such a machine is deterministic and complete. A FSM is called complete if for each state s of M, there is a transition from s with input symbol a, for every $a \in X$. A deterministic FSM does not allow two different transitions going out of the same state with identical input symbols.

Successive applications of the transition function δ give rise to the extended transition function $\widehat{\delta} : X^\star \times S \to S$, defined by

$$\begin{array}{lcl} \widehat{\delta}(\epsilon, s) & = & s, \\ \widehat{\delta}(a\rho, s) & = & \widehat{\delta}(\rho, \delta(a, s)), \text{where } a \in X \text{ and } \rho \in X^\star. \end{array}$$

Here, ϵ will denote the empty word. For convenience, if $\widehat{\delta}(\rho, s_1) = s_2$ we also write $s_1 \xrightarrow{\rho} s_2$.

We extend λ to $\widehat{\lambda} : X^\star \times S \to Y^\star$ thus

$$\begin{array}{lcl} \widehat{\lambda}(\epsilon, s) & = & \epsilon, \\ \widehat{\lambda}(a\rho, s) & = & \lambda(a, s)\widehat{\lambda}(\rho, \delta(a, s)), \text{with } a \in X, \rho \in X^\star. \end{array}$$

Henceforth, unless mention to the contrary, we will assume that M and M' denote FSMs in the form $M = (X, Y, S, s_0, \delta, \lambda)$ and $M' = (X, Y', S', s'_0, \delta', \lambda')$. Note that M and M' have the same input alphabet.

The reachability notion expresses the idea of starting at the initial state, traversing some transitions, and reaching a target state.

Definition 1 *A state s in a FSM M is reachable if and only if there exists $\rho \in X^\star$ such that $\widehat{\delta}(\rho, s_0) = s$.* ∎

We also say that $\widehat{\lambda}(\rho, s)$ is the behavior of M from state s over the input sequence ρ. The behavior of M over ρ is simply the behavior of M from s_0 over ρ. A sequence ρ distinguishes two states s_1 and s_2 of M if ρ gives distinct behaviors for s_1 and s_2, that is, if $\widehat{\lambda}(\rho, s_1) \neq \widehat{\lambda}(\rho, s_2)$.

2.2 FSM-based testing

Here, we briefly describe the basic W-method, which can be used for test generation using FSM models. We also briefly describe the related W_p and the HSI methods.

The W-method The objective is to verify whether implementation models conform to a specification model, as characterized by the behavior responses generated by external stimuli [3].

Basically, the application of this method consists in two main steps, given a specification FSM M and an implementation FSM M': (i) test sequences generation, based on M; and (ii) application of each test sequence to M and M', followed by a comparison of their respective behaviors.

The technique uses characterization sets of M in order to obtain a complete set of test case sequences. A characterization set, loosely speaking, can distinguish every pair of machine states (see Section 6). Let W be a characterization set for M. In order to obtain the test sequences, the W-method prefixes the sequences in W with certain sequences from $X^\star$, thus obtaining a set Z containing extended sequences. Furthermore, the method also computes a cover set P for M. Basically, applying sequences from P one can traverse any edge of M. The desired set of test sequences is the product PZ. More details to be presented in the sequel.

The W_p-method A related technique, the so called W_p-method [6], can potentially reduce the total length of the test sequences generated by the basic W-method. Again, let W be a characterization set for the specification model, M. For each state s_i of M, an identification subset $W_i \subseteq W$ is obtained. The idea is that for each state s_j of M, with $s_i \neq s_j$, there exists an input sequence $\rho_j \in W_i$ such that s_i and s_j are distinguishable by ρ_j, and no other proper subset of W_i has this property.

Then, a checking sequence for each state is prefixed to all sequences in the corresponding identification set. A checking sequence for a given state is simply an input sequence reaching that state, when starting at the initial state. It is proven [6] that the length of the resulting test sequences may be shorter, compared to those sequences obtained using the complete PZ concatenation of the basic W-method.

The HSI-method The HSI-method [16] uses the notion of trace-inclusion and a quasi-equivalence relation to verify conformance between partial non-deterministic FSM implementations and a given FSM specification. For that, so called harmonized sate identification sets are used instead of the identification subsets used in the W_p-method. Whereas identification sets fixed the sequences associated with a specific state s_i, a harmonized state identification set D_i, is constructed by taking prefixes of a characterization set W, but now allowing the reuse of a same prefix for different states. Distinguishing sequences for states are then taken from the intersection of D_i-sets. The discussion in [16] affirms that shorter sequences can be found to distinguish every pair of states in M.

3 Transition Covers

Let M be a FSM. A cover set $P \subseteq X^\star$ is required to exercise every transition in M, *i.e.*, for every transition

$\delta(a, s) = r$ in M there must be $\rho, \rho a \in P$ such that $\widehat{\delta}(\rho, s_0) = s$. In this way, we can obtain a behavior of M that reaches state s, and terminates by traversing the specific edge from s to r, labeled by a.

The cover set notion is formalized next.

Definition 2 *A set of input sequences $C \subseteq X^\star$ is a cover set for a FSM M if for every pair of states $s, r \in S$ and every input symbol $a \in X$, with $\delta(a, s) = r$, there exist $\rho, \rho a \in C$ such that $\widehat{\delta}(\rho, s_0) = s$.* ∎

A cover set can be obtained by constructing a labeled tree for M. A labeled tree is a system $T = (N, A, l_v, l_e)$, where N is a set of nodes, A is the set of edges, and $l_v : N \to S$ and $l_e : A \to X$ are labeling functions of nodes and edges, respectively. The nodes in the tree will be labeled by states of M and edges will be labeled by symbols from X.

Construction 3 *A labeled tree for M, $T = (N, A, l_v, l_e)$, can be constructed as follows:*

1. *Initiate with $N = \{n_0\}$, $A = \emptyset$, $l_v(n_0) = s_0$ and $l_e = \emptyset$, where s_0 is the initial state of M and n_0 is the root of T. We say that n_0 has level zero in T.*
2. *Inductively, suppose T is already constructed up to level $k \geq 0$. Level $k + 1$ is constructed by inspecting of nodes in level k from left to right:*
   (a) *let $n \in N$ be the next node to be inspected.*
   (b) *if there already exists $m \in N$ with $l_v(m) = l_v(n)$, and m is at some level $l < k$ in T, then node n is ignored, and we take the next node at level k. Otherwise, for every input $a \in X$ and every $r \in S$ with $r = \delta(a, l_v(n))$, we add a new node n' to N, a new edge (n, n') to A, and define $l_v(n') = r$ and $l_e(n, n') = a$. We then proceed to the next node in level k.*
3. *Step 2 is repeated if new nodes were added to T in the last iteration; otherwise, T is completed.* ∎

The process will always terminate since the set of states in M is finite. Depending on how the symbols from X are selected, different trees can be obtained (see step 2b in Construction 3).

The next definition shows how to construct a required cover set.

Definition 4 *Let T be a labeled tree for M. The set P_T is defined by all words $\alpha \in X^\star$ which label paths in T, starting at the root.* ∎

Note that $\epsilon \in P_T$. When T is clear from the context, we will use the simplified notation P instead of P_T.

We can now show that P_T, from Definition 4, is a cover set for machine M. Before that, we need a property of labelled trees.

Lemma 5 *Let $T = (N, A, l_v, l_e)$ be a labeled tree for a FSM M, as given by Construction 3. Let P_T be the set obtained as in Definition 4. Let $\rho \in X^\star$ and $s \in S$ be such that $\widehat{\delta}(\rho, s_0) = s$. Then, there exists a node $n \in N$ with $l_v(n) = s$. Furthermore, there exists a sequence $\alpha \in P_T$ with $\widehat{\delta}(\alpha, s_0) = s$ and such that for every edge $\delta(a, s) = r$ we have $\alpha a \in P_T$.*

Proof Directly from Construction 3. Details in [2]. ∎

Now we can enunciate the cover set property.

Corollary 6 *Let $T = (N, A, l_v, l_e)$ be the labeled tree for a FSM M, as given by Construction 3. Let P_T be the set obtained as in Definition 4. If every state of M is reachable, then the set $P_T \subseteq X^\star$ is a cover set for M.*

Proof Let $\delta(a, r) = s$ be an edge. As r is reachable, we have $\widehat{\delta}(\rho, s_0) = r$, for some $\rho \in X^\star$. By Lemma 5, we have α, $\alpha a \in P_T$ with $\widehat{\delta}(\alpha, s_0) = r$, for some $\alpha \in X^\star$. Thus, P_T is a cover set for M. ∎

4 Equivalences and Stratified Families

This section deals with state equivalence relations induced by the transition functions of the extended machines. The next definition exposes those notions in a general context.

Definition 7 *Let M and M' be two FSMs over the same input alphabet, X, and let s and s' be states of M and M', respectively.*

1. *Let $\rho \in X^\star$. We say that s is ρ-equivalent to s' if $\widehat{\lambda}(\rho, s) = \widehat{\lambda}'(\rho, s')$. In this case, we write $s \approx_\rho s'$. Otherwise, s and s' are ρ-distinguishable and we write $s \not\approx_\rho s'$.*
2. *Let $K \subseteq X^\star$. We say that s is K-equivalent to s' if s is ρ-equivalent to s', for every $\rho \in K$. In this case, we write $s \approx_K s'$. Otherwise, s and s' are K-distinguishable and we write $s \not\approx_K s'$.*
3. *Let $k \geq 0$. We say that s is k-equivalent to s' if s is X^k-equivalent to s'. Otherwise, s and s' are k-distinguishable. We write, respectively, $s \approx_k s'$ and $s \not\approx_k s'$.*
4. *State s is equivalent to s' if s is k-equivalent to s', for every $k \geq 0$. Otherwise, s and s' are distinguishable. We write, respectively, $s \approx s'$ and $s \not\approx s'$.* ∎

We will avoid overloading the notation by indicating M and M' explicitly, *e.g.*, in the form $\approx_k^{M,M'}$, since both machines will always be clear from the context. Definition 7, obviously, also applies when M and M' are the same machine.

In this case, it is easy to verify that all relations defined above are, in fact, equivalence relations over the state set of the machine. Hence, each such equivalence relation $\approx_Z$ gives rise to a partition $[Z]$ of the state set S.

Definition 8 *Let M be a FSM. The index of M, ι_M, is the number of equivalence classes induced by the $\approx$ relation over the states of M.* ∎

Clearly, we will always have $1 \leq \iota_M \leq |S|$, where S is the state set of M.

The next lemma gathers some simple observations.

Lemma 9 *Let M and M' be two FSMs with states s and s', respectively.*

1. *Let $K \subseteq X^\star$. If $s \approx_K s'$, then $s \approx_L s'$, for every L with $L \subseteq K$. On the other hand, if $s \not\approx_K s'$, then $s \not\approx_L s'$, for every L with $K \subseteq L$.*
2. *Let $k \geq 0$. If $s \approx_k s'$ then $s \approx_l s'$ for every l with $l \leq k$. On the other hand, if $s \not\approx_k s'$, then $s \not\approx_l s'$, for every l with $l \geq k$.*
3. *Let $K, L \subseteq X^\star$. If $s \not\approx_K s'$, then $s \not\approx_{KL} s'$, for every $L \neq \emptyset$.*

Proof Trivial. ∎

In the sequel, we will be considering specific sets of input sequences.

Definition 10 *Let $Z_i \subseteq X^\star$, $i \geq 0$, where X is an alphabet. We say that $\{Z_i\}_{i\geq 0}$ is a stratified family over X if*

1. *$Z_0 \neq \emptyset$; and*
2. *$(X \cup \{\epsilon\})Z_i = Z_{i+1}$, for every $i \geq 0$.* ∎

It is easy to see that these properties are independent of each other.

Another characterization for stratification is given as follows.

Proposition 11 *Let $Z_i \subseteq X^\star$, $i \geq 0$, where X is an alphabet and with $Z_0 \neq \emptyset$. Then, the family $\{Z_i\}_{i\geq 0}$ is stratified if and only if $Z_k = \bigcup_{j=0}^{k} X^j Z_0$ for every $k \geq 0$.*

Proof Details can be found in [2]. ∎

The next result guarantees that certain sequences always have continuations in some of the Z_k sets.

Lemma 12 *Let $\{Z_i\}_{i\geq 0}$ be a stratified family over X and let $k \geq 0$. Then*

1. *$Z_k \subseteq Z_j$, for every $j \geq k$; and*
2. *For every $\alpha \in X^j$, with $0 \leq j \leq k$, there exists $\beta \in X^\star$ such that $\alpha\beta \in Z_k$.*

Proof From Proposition 11, we deduce that $Z_i \subseteq Z_{i+1}$, for every $i \geq 0$. A simple induction establishes item (1). For item (2), since $Z_0 \neq \emptyset$, we take $\gamma \in Z_0$. Since $j \leq k$, we take $\sigma \in X^{k-j}$. Hence, $\alpha\sigma\gamma \in X^k Z_0$. From Proposition 11 we conclude $\alpha\sigma\gamma \in Z^k$. ∎

Let M be a FSM and let $Z \subseteq X^\star$ be a set of input sequences. We indicate by $[Z]$ the partition induced by Z (see observation after Definition 7) over the states of M, i.e, $s \approx_Z r$ if and only if $s, r \in w$, for some $w \in [Z]$. Let $[Z_1]$ and $[Z_2]$ be two partitions over S. Then we say that $[Z_2]$ *refines* $[Z_1]$ if and only if for all $w_2 \in [Z_2]$ there exists some $w_1 \in [Z_1]$ such that $w_2 \subseteq w_1$.

The next result expresses properties of these partitions.

Lemma 13 *Let $\{Z_i\}_{i\geq 0}$ be a stratified family over the alphabet X of a FSM M. Then*

1. *$[Z_{i+1}]$ refines $[Z_i]$, for every $i \geq 0$; and*
2. *if $|[Z_k]| = |[Z_{k+1}]|$ for some $k \geq 0$, then we must have $[Z_k] = [Z_{k+1}] = [Z_{k+2}]$.*

Proof We show each item, in turn.

For item (1), assume that it does not hold for some $i \geq 0$. Then we will have states s and r such that $s \approx_{Z_{i+1}} r$ and $s \not\approx_{Z_i} r$. From Lemma 9(1) and Lemma 12(1) we deduce $s \not\approx_{Z_{i+1}} r$, a contradiction.

Now we verify item (2). From item (1), we know that $[Z_{k+1}]$ refines $[Z_k]$. Then $[Z_k] = [Z_{k+1}]$, otherwise we would have $|[Z_k]| < |[Z_{k+1}]|$. Again continuing by contradiction, assume that $[Z_{k+1}] \neq [Z_{k+2}]$. Since $[Z_{k+2}]$ refines $[Z_{k+1}]$, we will have states r and s such that $s \not\approx_{Z_{k+2}} r$ and $s \approx_{Z_{k+1}} r$. Hence, we obtain $\rho \in Z_{k+2}$, with $\rho = a\beta$ and $a \in X$, and such that $s \not\approx_{a\beta} r$. We also conclude that $a\beta \notin Z_{k+1}$, otherwise we would have the contradiction $s \not\approx_{Z_{k+1}} r$. Therefore, from Definition 10(2), we deduce $a\beta \in XZ_{k+1}$, and so, $\beta \in Z_{k+1}$.

Let $s_1, r_1 \in S$ with $s_1 = \delta(a, s)$, $r_1 = \delta(a, r)$. If $s_1 \not\approx_{Z_{k+1}} r_1$ then $s_1 \not\approx_{Z_k} r_1$, because we already know that $[Z_k] = [Z_{k+1}]$. Hence, we would have $\gamma \in Z_k$ with $\widehat{\lambda}(\gamma, s_1) \neq \widehat{\lambda}(\gamma, r_1)$. From Definition 10(1) we have $XZ_k \subseteq Z_{k+1}$, and then $a\gamma \in Z_{k+1}$. But,

$$\begin{aligned} \widehat{\lambda}(a\gamma, s) &= \lambda(a, s)\widehat{\lambda}(\gamma, s_1) \\ \widehat{\lambda}(a\gamma, r) &= \lambda(a, r)\widehat{\lambda}(\gamma, r_1). \end{aligned}$$

Then we have $\widehat{\lambda}(a\gamma, s) \neq \widehat{\lambda}(a\gamma, r)$, thus forcing the contradiction $s \not\approx_{Z_{k+1}} r$. We conclude that $s_1 \approx_{Z_{k+1}} r_1$.

Since $\beta \in Z_{k+1}$, we deduce $s_1 \approx_\beta r_1$. Again,

$$\begin{aligned} \widehat{\lambda}(a\beta, s) &= \lambda(a, s)\widehat{\lambda}(\beta, s_1) \\ \widehat{\lambda}(a\beta, r) &= \lambda(a, r)\widehat{\lambda}(\beta, r_1), \end{aligned}$$

and, since we already have $\widehat{\lambda}(\rho, s) \neq \widehat{\lambda}(\rho, r)$, we conclude that $\lambda(a, s) \neq \lambda(a, r)$. From $a \in X$ and Lemma 12(2) we infer $\sigma \in X^\star$ with $a\sigma \in Z_{k+1}$. Hence, we have $s \not\approx_{a\sigma} r$, contradicting $s \approx_{Z_{k+1}} r$. ∎

The next result gives the equality of successive partitions.

Corollary 14 *Let $\{Z_i\}_{i\geq 0}$ be a stratified family over the input alphabet X of a FSM M. If $|[Z_k]| = |[Z_{k+1}]|$ for some $k \geq 0$, then $[Z_k] = [Z_{k+l}]$ for every $l \geq 0$.*

Proof When $l = 0$, the result is immediate. When $l = 1$ or $l = 2$, the result follows directly from Lemma 13(2). Assume the result holds for every j, $0 \leq j \leq l$, with $l \geq 2$. We want to show that the result holds for $l + 1$. From the induction, we have $[Z_k] = [Z_{k+l}]$ and $[Z_k] = [Z_{k+l-1}]$. Hence, $[Z_{k+l-1}] = [Z_{k+l}]$. Using Lemma 13(2), we obtain $[Z_{k+l-1}] = [Z_{k+l}] = [Z_{k+l+1}]$. Hence, $[Z_k] = [Z_{k+l+1}]$, as required. ∎

Now let M be a FSM with m states. Suppose we have a stratified family for X, $\{Z_i\}_{i\geq 0}$, in which Z_0 partitions the states of M in $n \leq m$ equivalence classes. We want to study the partitions over states of M induced by the Z_i sets, for $i \geq 0$. The next lemma establishes the basic result.

Lemma 15 *Let M be a FSM with index m. Let $\{Z_i\}_{i\geq 0}$ be a stratified family for X such that Z_0 partitions the states of M in at least $n \leq m$ equivalence classes. Then $|[Z_i]| \geq n + i$, for every i, with $0 \leq i \leq m - n$.*

Proof When $i = 0$ we have $n + i = n$ and, from the hypothesis, $|[Z_0]| \geq n$, establishing the base. Assume the result for every j, $0 \leq j \leq i$, with $i < m - n$. We are going to show that the result holds for $i + 1$. If $|[Z_i]| \geq n + i + 1$ then $|[Z_{i+1}]| \geq n + i + 1$ (from Lemma 13(1)), and the induction is extended in this case.

Now, let $|[Z_i]| < n+i+1$. From the induction hypothesis we conclude that $|[Z_i]| = n+i$. Since $m \geq n+i+1$ is the index of M, there exist nonequivalent states in M, r and s, with $r \approx_{Z_i} s$. Then, $s \not\approx_{X^k} r$, for some $k \geq 0$ (see Definition 7). From Lemma 12(2), we conclude $s \not\approx_{Z_k} r$. If $k \leq i$, Lemma 12(1) would force $Z_k \subseteq Z_i$. Using Lemma 9(1) we would have $s \not\approx_{Z_i} r$, a contradiction. Hence, $k > i$.

If $|[Z_i]| = |[Z_{i+1}]|$ then, by Corollary 14, we get $Z_i = Z_k$, forcing again the contradiction $s \not\approx_{Z_i} r$. Since $[Z_{i+1}]$ refines $[Z_i]$, we can not have $|[Z_{i+1}]| < |[Z_i]|$. We conclude that $|[Z_{i+1}]| > |[Z_i]|$. But, since $|[Z_i]| = n + i$, we deduce the result desired, that is, $|[Z_{i+1}]| \geq n + i + 1$. ∎

Using this result, it will be easy to confirm that some $Z \in \{Z_i\}_{i\geq 0}$ will distinguish every pair of nonequivalent states.

Corollary 16 *Let M be a FSM with index m. Let $\{Z_i\}_{i\geq 0}$ be a stratified family for X such that Z_0 partitions the states of M in at least $n \leq m$ equivalence classes. Then Z_{m-n} will distinguish every pair of nonequivalent states of M.*

Proof From Lemma 15, it follows that $|[Z_{m-n}]| \geq n + (m - n) = m$. Since $[Z_{m-n}]$ is the partition induced by Z_{m-n}, we conclude that Z_{m-n} partitions states of M in m classes. Since M has index m, we conclude that Z_{m-n} will distinguish every pair of nonequivalent states of M. ∎

5 A m-complete Test Suite

Let M and M' be two FSMs operating over the same alphabet X. Machine M represents a specification and M' represents a possible implementation. We want to obtain a set $K \subseteq X^\star$ such that $s_0 \not\approx s'_0$ if and only if $s_0 \not\approx_K s'_0$. Such a set K is a m-complete test suite, where m is an upper bound on the index of M'. Given K, if we want to test whether M and M' have distinct behaviors, it is enough to apply the sequences in K to both machines and compare the corresponding output sequences.

We obtain the required set by combining a cover set for M with a stratified family for M'. The next lemma establishes an auxiliary result.

Lemma 17 *Let M and M' be two FSMs operating over the same input alphabet, X. Assume that M' has index m and that P is a cover set for M. Let $Z \subseteq X^\star$ be nonempty and such that Z partitions the states of M' in at least m equivalence classes. If $s_0 \approx_{PZ} s'_0$ and $s_0 \not\approx s'_0$, then there exist $\gamma \in X^\star$, $s \in S$, $s' \in S'$ such that $\widehat{\delta}(\gamma, s_0) = s$, $\widehat{\delta}'(\gamma, s'_0) = s'$ and $s \not\approx_Z s'$.*

Proof This proof can be found in [2]. ∎

Now we are in a position to enunciate the result which will give us the capability of testing two machines for equivalence.

Theorem 18 *Let M and M' be two FSMs operating over the same input alphabet, X. Assume that M' has index m and that P is a cover set for M. Let $Z \subseteq X^\star$ be nonempty and such that Z partitions states of M' in at least m equivalence classes. Then, $s_0 \approx s'_0$ if and only if $s_0 \approx_{PZ} s'_0$.*

Proof If $s_0 \approx s'_0$ then, trivially, $s_0 \approx_{PZ} s'_0$.

For the opposite direction, assume $s_0 \approx_{PZ} s'_0$. For the sake of contradiction, assume $s_0 \not\approx s'_0$. From Lemma 17, we obtain $\beta \in X^\star$, $s \in S$ and $s' \in S'$ with $\widehat{\delta}(\beta, s_0) = s$, $\widehat{\delta}'(\beta, s'_0) = s'$, and $s \not\approx_Z s'$. We can assume, without loss of generality, that $|\beta|$ is minimal. If $\beta = \epsilon$, we would have $s = s_0$ and $s' = s'_0$, and then $s_0 \not\approx_Z s'_0$. But, since $\epsilon \in P$, this would force the contradiction $s_0 \not\approx_{PZ} s'_0$. We conclude

that $\beta = \alpha a$, with $a \in X$. Let $r \in S$ and $r' \in S'$ with $\widehat{\delta}(\alpha, s_0) = r$, $\widehat{\delta'}(\alpha, s'_0) = r'$, $\delta(a, r) = s$ and $\delta'(a, r') = s'$. Using the minimality of $|\beta|$ we have $r \approx_Z r'$.

On the other hand, since P is a cover set for M, from the edge $\delta(a, r) = s$ we obtain $\rho \in P$ and $\rho a \in P$ with $\widehat{\delta}(\rho, s_0) = r$. Let $r'' \in S'$ with $\widehat{\delta'}(\rho, s'_0) = r''$. If we had $r \not\approx_Z r''$, we would obtain $\gamma \in Z$ with $\widehat{\lambda}(\gamma, r) \neq \widehat{\lambda'}(\gamma, r'')$. But then

$$\begin{aligned} \widehat{\lambda}(\rho\gamma, s_0) &= \widehat{\lambda}(\rho, s_0)\widehat{\lambda}(\gamma, r) \quad \text{and} \\ \widehat{\lambda'}(\rho\gamma, s'_0) &= \widehat{\lambda'}(\rho, s'_0)\widehat{\lambda'}(\gamma, r''). \end{aligned}$$

Hence, $\widehat{\lambda}(\rho\gamma, s_0) \neq \widehat{\lambda'}(\rho\gamma, s'_0)$, giving the contradiction $s_0 \not\approx_{\rho\gamma} s'_0$ with $\rho\gamma \in PZ$. We conclude that $r \approx_Z r''$.

Since we already have $r \approx_Z r'$, we obtain $r' \approx_Z r''$. Since Z partitions the states of M' in m classes and m is the index of M', we conclude that $r' \approx r''$. Now, from $s \not\approx_Z s'$, we obtain $\sigma \in Z$ with $\widehat{\lambda}(\sigma, s) \neq \widehat{\lambda'}(\sigma, s')$. But,

$$\begin{aligned} \widehat{\lambda}(\rho a\sigma, s_0) &= \widehat{\lambda}(\rho, s_0)\lambda(a, r)\widehat{\lambda}(\sigma, s) \quad \text{and} \\ \widehat{\lambda'}(\rho a\sigma, s'_0) &= \widehat{\lambda'}(\rho, s'_0)\widehat{\lambda'}(a\sigma, r'') \\ &= \widehat{\lambda'}(\rho, s'_0)\widehat{\lambda'}(a\sigma, r') \\ &= \widehat{\lambda'}(\rho, s'_0)\lambda'(a, r')\widehat{\lambda'}(\sigma, s'). \end{aligned}$$

Then, $\widehat{\lambda}(\rho a\sigma, s_0) \neq \widehat{\lambda'}(\rho a\sigma, s'_0)$. But $\rho a\sigma \in PZ$ and we would have $s_0 \not\approx_{PZ} s'_0$, contradicting the hypothesis. This concludes the proof. ∎

Combining the previous results, we have the following corollary, useful to determine whether two FSMs have distinguishing behaviors.

Corollary 19 *Let M and M' two FSMs operating over the same input alphabet, X. Assume that M' has index m. Assume also that P is a cover set for M, that $R \subseteq X^\star$ is nonempty and that it partitions the states of M' in at least $n \leq m$ equivalence classes. Then, s_0 and s'_0 are equivalent if and only if s_0 and s'_0 are PZ-equivalent, where $Z = \bigcup_{i=0}^{m-n} X^i R$.*

Proof Let $Z_k = \bigcup_{i=0}^{k} X^i R$, $k \geq 0$. From Proposition 11 we have that such family $\{Z_k\}_{k\geq 0}$ is stratified. From Corollary 16 we conclude that Z distinguishes every pair of nonequivalent states of M'. Then the result follows directly from Theorem 18. ∎

6 Characterization Sets

From the previous corollary, it might appear that Z and M are independent, since the only hypothesis involving M, in that corollary, is that P is a cover set for M. But, in fact, there is a relationship between Z and M. Before we expose the relationship between Z and M, we need another auxiliary result.

Lemma 20 *Let M and M' be two FSMs operating over the same input alphabet, X. Assume that all states of M are reachable and that $s_0 \approx s'_0$. Let $Z \subseteq X^\star$ be a set partitioning the states of M' in m equivalence classes, where m is the index of M'. Then Z distinguishes every pair of nonequivalent states of M.*

Proof Let $s_1, s_2 \in S$ with $s_1 \not\approx s_2$ and assume $s_1 \approx_Z s_2$. Since all states of M are reachable, we have $\rho_1, \rho_2 \in X^\star$ such that $\widehat{\delta}(\rho_i, s_0) = s_i$, with $i = 1, 2$. In M' we would have some $s'_1, s'_2 \in S'$ and with $\widehat{\delta'}(\rho_i, s'_0) = s'_i$, where $i = 1, 2$.

Now let $\beta \in Z$. We have,

$$\begin{aligned} \widehat{\lambda}(\rho_2\beta, s_0) &= \widehat{\lambda}(\rho_2, s_0)\widehat{\lambda}(\beta, s_2) \\ \widehat{\lambda'}(\rho_2\beta, s'_0) &= \widehat{\lambda'}(\rho_2, s'_0)\widehat{\lambda'}(\beta, s'_2) \end{aligned}$$

and, since $s_0 \approx s'_0$, we obtain $\widehat{\lambda}(\beta, s_2) = \widehat{\lambda'}(\beta, s'_2)$ and $\widehat{\lambda}(\rho_2, s_0) = \widehat{\lambda'}(\rho_2, s'_0)$. Since β is arbitrary, we conclude that $s_2 \approx_Z s'_2$.

Similarly,

$$\begin{aligned} \widehat{\lambda}(\rho_1\beta, s_0) &= \widehat{\lambda}(\rho_1, s_0)\widehat{\lambda}(\beta, s_1) \\ \widehat{\lambda'}(\rho_1\beta, s'_0) &= \widehat{\lambda'}(\rho_1, s'_0)\widehat{\lambda'}(\beta, s'_1), \end{aligned}$$

and we conclude that $s_1 \approx_Z s'_1$, together with $\widehat{\lambda}(\rho_1, s_0) = \widehat{\lambda'}(\rho_1, s'_0)$.

Putting it together, and knowing that $s_1 \approx_Z s_2$, we obtain $s_1 \approx_Z s'_2$ and also $s_2 \approx_Z s'_1$. Hence, $s'_1 \approx_Z s'_2$. But s'_1 and s'_2 are states of M' and so the hypothesis over Z gives $s'_1 \approx s'_2$.

On the other hand, since $s_1 \not\approx s_2$, we obtain $\sigma \in X^\star$ such that $\widehat{\lambda}(\sigma, s_1) \neq \widehat{\lambda}(\sigma, s_2)$. Now,

$$\begin{aligned} \widehat{\lambda}(\rho_1\sigma, s_0) &= \widehat{\lambda}(\rho_1, s_0)\widehat{\lambda}(\sigma, s_1) \\ \widehat{\lambda'}(\rho_1\sigma, s'_0) &= \widehat{\lambda'}(\rho_1, s'_0)\widehat{\lambda'}(\sigma, s'_1). \end{aligned}$$

Hence, from $\widehat{\lambda}(\rho_1\sigma, s_0) = \widehat{\lambda'}(\rho_1\sigma, s'_0)$ and $\widehat{\lambda}(\rho_1, s_0) = \widehat{\lambda'}(\rho_1, s'_0)$, we deduce $\widehat{\lambda}(\sigma, s_1) = \widehat{\lambda'}(\sigma, s'_1)$.

Similarly,

$$\begin{aligned} \widehat{\lambda}(\rho_2\sigma, s_0) &= \widehat{\lambda}(\rho_2, s_0)\widehat{\lambda}(\sigma, s_2) \\ \widehat{\lambda'}(\rho_2\sigma, s'_0) &= \widehat{\lambda'}(\rho_2, s'_0)\widehat{\lambda'}(\sigma, s'_2), \end{aligned}$$

and then $\widehat{\lambda}(\sigma, s_2) = \widehat{\lambda'}(\sigma, s'_2)$. However, since we already know that $s'_1 \approx s'_2$ and this leads to the contradiction $\widehat{\lambda}(\sigma, s_1) = \widehat{\lambda}(\sigma, s_2)$. This shows that the initial hypothesis was false. Hence, whenever $s_1 \not\approx s_2$ holds we must also have $s_1 \not\approx_Z s_2$, establishing the result. ∎

A set in these conditions is called a characterization set of M.

Definition 21 *Let M be a FSM and W a set of input sequences. W is a characterization set for M if W distinguishes any pair of nonequivalent states of M.* ∎

The required relation between M and Z says that Z is a characterization set of M, under certain hypothesis.

Theorem 22 *Let M and M' be two FSMs operating over the same input alphabet, X. Assume that M' has index m and that P is a cover set for M. Assume also that $W \subseteq X^\star$ is nonempty and partitions the states of M' in at least $n \leq m$ equivalence classes. If $s_0 \approx_{PZ} s'_0$ then $Z = \bigcup_{i=0}^{m-n} X^i W$ is a characterization set for M.*

Proof From Proposition 11 and from Corollary 16 we conclude that Z distinguishes every pair of nonequivalent states of M'. Since P is cover set for M, we conclude that every state of M is reachable. From $s_0 \approx_{PZ} s'_0$, together with Corollary 19, we deduce $s_0 \approx s'_0$. Now we can use Lemma 20 and obtain that Z distinguishes every pair of nonequivalent states of M. From Definition 21, Z is a characterization set for M. ∎

It is also easy to see that the reverse does not hold. For that, let M and M' be two FSMs. It is clear that $W = X^\star$ partitions the states of M and M' in the maximum number of equivalence classes. In this case, we will have $Z = W = X^\star$ and, obviously, Z is a characterization set for M and M'. But it is not the case that we will always have $s_0 \approx s'_0$, as it is easy to construct a counter-example.

Next result shows that, under relaxed conditions, when two FSMs are equivalents both must have the same index.

Theorem 23 *Let M and M' be two FSMs operating over the same input alphabet, X. Let n and n' be the index of M and M', respectively. Assume that all states from both FSMs are reachable. If $s_0 \approx s'_0$ then $n = n'$.*

Proof For the sake of contradiction, and without loss generality, we will assume $n < n'$.

Let $s'_i \in S'$, $1 \leq i \leq n'$, be states from each one of n' equivalence classes induced by $\approx$ in S'. Since all states of M' are reachable, we obtain $\rho_i \in X^\star$ with $\widehat{\delta}'(\rho_i, s'_0) = s'_i$, $1 \leq i \leq n'$. In M, we will have some $s_i \in S$ such that $\widehat{\delta}(\rho_i, s_0) = s_i$, $1 \leq i \leq n'$. Since $n < n'$, without loss generality, we can say that $s_1 \approx s_2$.

Take any $z \in X^\star$. We have

$$\begin{aligned} \widehat{\lambda}(\rho_1 z, s_0) &= \widehat{\lambda}(\rho_1, s_0)\widehat{\lambda}(z, s_1) \quad \text{and} \\ \widehat{\lambda}'(\rho_1 z, s'_0) &= \widehat{\lambda}'(\rho_1, s'_0)\widehat{\lambda}'(z, s'_1). \end{aligned}$$

Since $s_0 \approx s'_0$, it follows that $\widehat{\lambda}(z, s_1) = \widehat{\lambda}'(z, s'_1)$. Similarly, $\widehat{\lambda}(z, s_2) = \widehat{\lambda}'(z, s'_2)$.

But since $s_1 \approx s_2$, we obtain $\widehat{\lambda}(z, s_1) = \widehat{\lambda}(z, s_2)$. Therefore, $\widehat{\lambda}'(z, s'_1) = \widehat{\lambda}'(z, s'_2)$. Since $z \in X^\star$ is arbitrary, we conclude that $s'_1 \approx s'_2$, a contradiction given that s'_1 and s'_2 are in distinct classes in M'.

Hence, we must have $n \geq n'$. Similarly, $n' \geq n$, and then $n = n'$. ∎

The same result indicates that when the $\approx$ relation induces a different number of equivalence classes in two FSMs, these machines can not be equivalent to each other (under the weak hypothesis of Theorem 23). On the other hand, it is simple to obtain two nonequivalent FSMs, in a such way that the $\approx$ relation induces the same number of equivalence classes in both machines. For details, see [2].

7 The W-method as a Particular Case

Consider the hypothesis of Theorem 22. We can show that W is a characterization set of M if n is the index of M and the behaviors of both machines must match.

Corollary 24 *Let M and M' be two FSMs operating over the same input alphabet, X, and assume that all states in M' are reachable. Assume further that M' has index m, that P is a cover set for M and that M has index n. Assume also that $W \subseteq X^\star$ is nonempty and partitions the states of M' in at least $n \leq m$ equivalence classes. If $s_0 \approx_{PZ} s'_0$, where $Z = \bigcup_{i=0}^{m-n} X^i W$, then $n = m$, $Z = W$ and W is a characterization set for M.*

Proof Since $s_0 \approx_{PZ} s'_0$, together with Corollary 19, we conclude that $s_0 \approx s'_0$. Next, we infer that $n = m$, from Theorem 23. Hence, $Z = W$. Therefore, by Theorem 22, W is a characterization set for M. ∎

When W is a characterization set for M we can guarantee the partitioning of M' in a number of classes at least equal to the index of M, if the machines are to be PZ-equivalent.

Lemma 25 *Let M and M' be two FSMs operating over the same input alphabet, X. Assume that M' has index m, that M has index n and that P is a cover set for M, with $n \leq m$. Assume also that $W \subseteq X^\star$ is a characterization set for M and that $s_0 \approx_{PZ} s'_0$, where $Z = \bigcup_{i=0}^{m-n} X^i W$. Then W partitions M' in at least n equivalence classes.*

Proof We know that M has n equivalence classes: Let $C_1, \ldots, C_n$ be these classes. Let $s_i \in C_i$ and $s_j \in C_j$, where $1 \leq i < j \leq n$. Then since W is a characterization set for M, we have $s_i \not\approx_W s_j$. Since P is cover set of M, we have $\widehat{\delta}(\rho, s_0) = s_i$, for some $\rho \in P$. We also know that $\widehat{\delta}'(\rho, s'_0) = s'_i$, for some s'_i of M'. Since $s_0 \approx_{PZ} s'_0$, we get $s_i \approx_Z s'_i$. Since $W \subseteq Z$, then $s_i \approx_W s'_i$. In the

same way, we have s'_j of M' with $s_j \approx_W s'_j$. Then we obtain $s'_i \not\approx_W s'_j$, otherwise $s_i \approx_W s_j$. We conclude that W partitions M' in at least $n \leq m$ equivalence classes. ∎

Now we can use Lemma 25 to show another version of Corollary 19, under the hypothesis that the basic set of input sequences is a characterization set for the specification.

Theorem 26 *Let M and M' be two FSMs operating over the same input alphabet, X. Assume that M' has index m, that P is a cover set for M and that M has index n, with $n \leq m$. Assume also that $W \subseteq X^\star$ is a characterization set for M and that $s_0 \approx_{PZ} s'_0$, where $Z = \bigcup_{i=0}^{m-n} X^i W$. Then $s_0 \approx s'_0$.*

Proof Assume $s_0 \approx_{PZ} s'_0$. Use Lemma 25 to show that W partitions M' in at least n classes. Now use Corollary 16 to show that Z partitions M' in m classes. Finally, use Theorem 18. ∎

The next result is the main postulate of the basic W-method, as given in [3].

Theorem 27 *Let M and M' be two FSMs operating over the same input alphabet, X. Assume that M' has index m, that P is a cover set for M and that M has index n, with $n \leq m$. Assume also that $W \subseteq X^\star$ is a characterization set for M. Then $s_0 \approx s'_0$ if and only if $s_0 \approx_{PZ} s'_0$, where $Z = \bigcup_{i=0}^{m-n} X^i W$.*

Proof If $s_0 \approx s'_0$, then $s_0 \approx_{PZ} s'_0$, trivially. For the other direction, use Theorem 26. ∎

In general, W need not be a characterization set for M (see Corollary 19). For the method to work, we need only guarantee that M' will be partitioned in at least n equivalence classes with $n \leq m$, where m is the index of M'. No relationship between W and M is needed. On the other hand, when using the basic W-method directly, we need to obtain a characterization set W for M, we need to know the index of M, and we also need to secure the relationship $n \leq m$. When W is not a characterization set for M, the method may fail, as shown by the following example.

Example 28 The alphabet of M and M' is $X = \{a, b, c\}$. See Figures 1 and 2. It is easy to see that M has index $n = 3$, because $s_1 \approx s_3$. The index of M' is $m = 3$ since $s'_1 \approx s'_3 \approx s'_4$. Hence $m = n$, and we would be left with $Z = W$ (see Theorem 27). Now take $W = \{\epsilon\}$. A cover set can be given by $P = \{\epsilon, aa, ab, ac, ba, bb, bc, ca, cb, cc\}$. Then, $PZ = PW = P$.

It is easy to see that M and M' are PZ-equivalent. But $s_0 \approx s'_0$ is not true. To see that, take $\alpha = bbb$. We have $\widehat{\lambda}(\alpha, s_0) = 100$ and $\widehat{\lambda}'(\alpha, s'_0) = 101$. Note how W induces only one equivalence class in M'. Therefore, clearly, W is not a characterization set for M. ∎

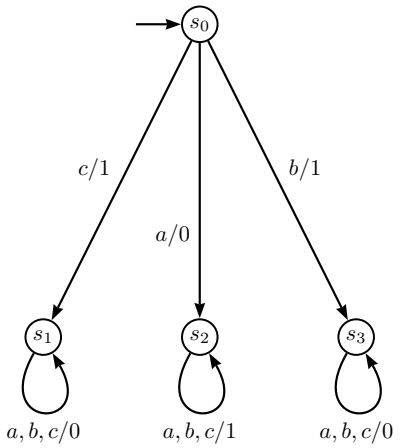


Figure 1. Specification M.

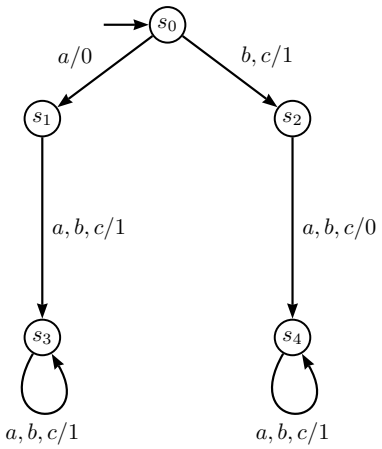


Figure 2. Implementation candidate M'.

In general, it would be important to devise a mechanism by which we could obtain the number of classes induced by W in M'. First, because in this case we might avoid calculating a characterization set for M when using our more general method. Secondly, we could potentially reduce the size of the sequences in Z, when W partitions M' in k classes, with $k > n$, given that $Z = \bigcup_{i=0}^{m-n} X^i W$.

8 The Generalized Test Generation Method

Algorithm 1 presents the generalized model-based test generation method. The input parameters are: M represents a system specification, M' is an implementation candidate for the specification M, R is any set of input sequences, n is a lower bound on the number of classes induced by R in M', and m is an upper bound on the index of M'. Thus, the method requires knowledge of a lower bound on the number n of equivalence classes induced by R in the implementa-

tion machine M', as well as an upper bound on the index m of M'. In an extreme case, once can set $n = 1$ and $m = |S'|$, that is, set m to the number of states in M'. Note that the implementation M' is given as a black box. So, we do not have access to its internal structure, and the parameters n and m must be estimated. As for the specification M, R may partition it in any number k of classes. Of course, if M and M' turn out to be equivalent, then they will have the same index and Z will, in fact, be a characterization set for both M and M'.

If the condition $n \leq m$ is secured and it turns out that M and M' are not equivalent, the algorithm produces a particular input sequence σ that is a witness to this fact, that is, M and M' display distinct behaviors over σ.

Algorithm 1: Generalized test generation algorithm.

Input: M, M', R, m, n
begin
  Obtain a cover set P for M;
  if $n \leq m$ **then**
    Compute $Z = \bigcup_{i=0}^{m-n} X^i R$;
    Compute PZ;
  else
    mesg: M and M' are not equivalent;
    return ;
  end
  foreach $\sigma \in PZ$ **do**
    Apply σ to M and to M';
    Obtain $y = \widehat{\lambda}(\sigma, s_0)$ and $y' = \widehat{\lambda'}(\sigma, s'_0)$;
    if $y \neq y'$ **then**
      mesg: M and M' are not equivalent;
      mesg: σ is an input witness;
      return ;
    end
  end
  mesg: M and M' are equivalents;
  return ;
end

In order to apply the basic W-method (see Theorem 27) some extra effort must be applied to compute the index of M as well as a characterization set for M.

In our proposal, we do not need characterization sets, nor is it necessary to inform the index of the specification machine M. On the other hand, practical information about M can aid in obtaining a good candidate for R. For example, based on the number of symbols in the input alphabet and on the number of states and transitions in M, some distinguishing sequences can be inserted into R. Then, it is easy to obtain the set Z using the notion of stratification. Clearly, after obtaining the concatenation PZ, we can use this product to verify conformance between the specification and several proposed implementations.

Note that the size of the PZ set depends on the algorithm used to obtain the cover set P. In fact, this algorithm is polynomial in the size of M (see Section 3). Furthermore, it depends on the choice of the set R and the bound m.

9 An Example

We apply the generalized algorithm to a simple example.

Example 29 Let a specification M be given as in Figure 3. Then M has $k = 4$ states, its input alphabet is $X = \{a, b\}$, its output alphabet is $Y = \{0, 1\}$, and its transition function is as depicted in the figure.

As we can see, some transitions over the input a produce either the output 0 or the output 1. Hence, there are at least two distinct classes. Now, if we use the sequences aa and ba there is a good chance that such sequences can distinguish other states as well. Therefore, we take $R = \{aa, ba\}$ and assume that R partitions M', an implementation candidate, in at least $n = 3$ equivalence classes. If we accept $m = 5$ as a maximum on the number of states in M', we have all input conditions for Algorithm 1 secured. Note that R is not

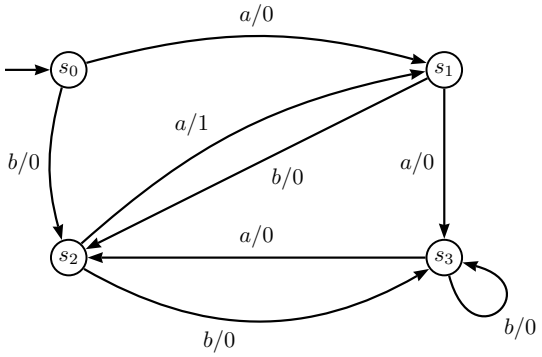


Figure 3. Machine specification M.

a characterization set for M because we have states s_0 and s_1 in the same equivalence class induced by R.

Next we calculate a cover set P for M. In the example, using the labeled tree construction (see Section 3) we get $P = \{\epsilon, a, b, aa, ab, ba, bb, aab, aaa\}$.

Now, with $m = 5$, $n = 3$ and $R = \{aa, ba\}$, we compute $Z = \bigcup_{i=0}^{m-n} X^i R$ and obtain

$$Z = \{aa, ba, aaa, aba, baa, bba, aaaa, \\ abaa, baaa, bbaa, aaba, abba, baba, bbba\}.$$

Then, the concatenation PZ will count 56 sequences.

10 Concluding Remarks

The Finite State Machine (FSM) model is well established and has been intensively investigated as a foundation

for the automatic generation of test cases. The W-method is a well known technique used to compute test sequences having FSMs as its basic formal model.

In this paper, our contribution is threefold. First, we generalized the basic W-method, avoiding the computation of characteristic sets and indexes. Secondly, we demonstrated in a clear way how the basic W-method follows from our generalized method. And finally, we presented detailed proofs of correctness both of our algorithm, as well as for the main tenets of the basic W-method, the latter being absent in the original work where it was introduced.

Note that some recent test generation methods, such as those presented in Section 2.2, have to calculate a characterization set of the specification, in the same way as the basic W-method. On the other hand, our method does not need characterization sets in order to generate test cases. We envisage that similar ideas can be used to extend and generalize other test case generation techniques, such as the W_p and HSI methods.

As future steps we plan to integrate the results presented in this paper with extensions of the basic FSM model, now also taking into account time constraints [1, 15].

References

[1] A. L. Bonifácio, A. V. Moura, A. da Silva Simão, and J. C. Maldonado. Towards deriving test sequences by model checking. *Electron. Notes Theor. Comput. Sci.*, 195:21–40, 2008.

[2] A. L. Bonifácio, A. V. Moura, and A. d. S. Simão. A generalized model-based test generation method. Technical Report IC-08-014, Instituto de Computação, Universidade Estadual de Campinas, Campinas, May 2008.

[3] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.

[4] S. J. Cunning and J. W. Rozenblit. Automating test generation for discrete event oriented embedded system s. *J. Intell. Robotics Syst.*, 41(2-3):87–112, 2005.

[5] R. Dorofeeva, K. El-Fakih, and N. Yevtushenko. An improved conformance testing method. In *FORTE*, pages 204–218, 2005.

[6] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, June 1991.

[7] A. Gargantini. Conformance testing. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 87–111. Springer-Verlag, 2005.

[8] A. Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill, New York, 1962.

[9] G. Gonenc. A method for the design of fault detection experiments. *IEEE Trans. Comput.*, 19(6):551–558, 1970.

[10] F. C. Hennie. Fault detecting experiments for sequential circuits. In *FOCS*, pages 95–110, 1964.

[11] R. M. Hierons. Separating sequence overlap for automated test sequence generation. *Automated Software Engg.*, 13(2):283–301, 2006.

[12] M. Krichen. State identification. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*, pages 87–111. Springer-Verlag, 2005.

[13] G. Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state ma chines using a generalized wp-method. *IEEE Trans. Softw. Eng.*, 20(2):149–162, 1994.

[14] B. Nielsen and A. Skou. Test generation for time critical systems: Tool and case study. *ecrts*, 00:0155, 2001.

[15] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in efsm testing. *IEEE Trans. Softw. Eng.*, 30(1):29–42, 2004.

[16] A. Petrenko and G. v. Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. In G. Luo, editor, *IWPTS '94: 7th IFIP WG 6.1 international workshop on Protocol test systems*, pages 95–110, London, UK, UK, 1995. Chapman & Hall, Ltd.

[17] A. Petrenko and N. Yevtushenko. Testing from partial deterministic fsm specifications. *IEEE Trans. Comput.*, 54(9):1154–1165, 2005.

[18] A. Rezaki and H. Ural. Construction of checking sequences based on characterization sets. *Computer Communications*, 18(12):911–920, 1995.

[19] D. Sidhu and T. Leung. Experience with test generation for real protocols. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 257–261, New York, NY, USA, 1988. ACM.

[20] D. P. Sidhu and T. kau Leung. Formal methods for protocol testing: A detailed study. *IEEE Trans. Softw. Eng.*, 15(4):413–426, 1989.

[21] J. Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 1996.

[22] J. Tretmans. Testing concurrent systems: A formal approach. In J. Baeten and S. Mauw, editors, *CONCUR '99: Proceedings of the 10th International Conference on Co ncurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65, London, UK, 1999. Springer-Verlag.

[23] H. Ural, X. Wu, and F. Zhang. On minimizing the lengths of checking sequences. *IEEE Trans. Comput.*, 46(1):93–99, 1997.