Teste de Software

Autor: Leandro César Prudente

`lecesar@ime.usp.br`

Orientador: Ana Cristina Vieira Melo

`acvm@ime.usp.br`

Departamento de Ciência da Computação

Instituto de Matemática e Estatística

Universidade de São Paulo

Sumário

1 Teste de Software

O desenvolvimento de sistemas de software envolve uma série de atividades de produção nas quais as oportunidades de injeção de falhas humanas são enormes[14]. As atividades relacionadas à Garantia de Qualidade de Software têm sido aplicadas ao longo do desenvolvimento do software, entre elas a atividade de teste. A atividade de teste[4] consiste de uma análise dinâmica do produto, sendo relevante para a identificação e eliminação de erros que persistem. Segundo Pressman[14], "A atividade de teste não pode mostrar a ausência de bugs, ela só pode mostrar se defeitos de software estão presentes".

1.1 Objetivos do Teste

Segundo Myers[5] " Teste de software é o processo de executar um programa com o objetivo de encontrar defeitos". Um teste bem sucedido é aquele que consegue determinar *casos de teste* para os quais o programa em teste falhe. Um *caso de teste* consiste na análise de um *dado de teste*[1] associado a um resultado esperado do processamento, segundo a especificação do software[5].

Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto. O caso de teste também ajuda a verificar se o software está se comportando segundo a sua especificação (formal ou informal). Muitos projetistas para ter certeza que seus códigos refletem seus modelos, testam seus softwares durante o desenvolvimento e refinam seus projetos e seus códigos baseados no resultado dos testes.

A atividade de teste tem sido apontada entre as mais complexa no desenvolvimento de software, pois uma pequena alteração numa função do sistema, com o objetivo de resolver um problema encontrado por um caso de teste, pode acarretar em alterações em todo o sistema. Desta forma, tem-se como objetivo principal projetar casos de teste que descubram sistematicamente diferentes classes de erros e que tenham custos de tempo e de esforço mínimos. O fato de não se encontrar erros, não significa necessariamente que o software não os possua, isso pode significar que o conjunto de casos de teste utilizado não é adequado para revelá-los. Para tentar reduzir o custo relacionado à atividade de teste é necessário fazer um planejamento conhecendo todos os processos de teste e com isso pode-se aplicar as técnicas e os critérios julgados relevantes em cada um dos processos.

A atividade de teste é vista como *destrutiva*, dado que tem como objetivo revelar os problemas existentes no produto. Mas do ponto de vista da Engenharia de Software, o teste é visto como uma tarefa

[1] Conjuntos de valores utilizados como entradas para o programa em uma determinada execução. Conhecido também como dados de entrada.

construtiva, pois tem como objetivo obter um software de melhor qualidade.

1.2 Conceitos Importantes

Antes de relatar os planejamentos e processos de teste, é relevante destacarmos algumas definições que são amplamente utilizadas na literatura de teste[5, 4]:

1. **Falha** - acontece quando uma saída incorreta é produzida com relação à especificação do programa;

2. **Defeito** - é uma deficiência no programa (um passo, um processo ou uma definição de dados incorretos) que pode provocar uma saída incorreta para algum dado de entrada;

3. **Erro** - é uma diferença existente entre o valor correto e o valor esperado para alguma variável na execução do programa. Ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro;

4. **Engano** - ação humana que produz um resultado incorreto.

1.3 Planejamento e Processo de teste

A atividade primordial de teste é executar um programa, fornecendo *dados de testes*, e comparar os valores produzidos com o resultado esperado. Caso a saída seja diferente do resultado esperado, tem-se uma falha de programa e para identificar e corrigir tal discrepância é necessário fazer uma depuração(*debugging*). Depuração não é teste, mas um processo que resulta na remoção e/ou correção dos erros. O processo de depuração[14] avalia os resultados do caso de teste e qualquer falta de correspondência entre o resultado esperado e o real é encontrada. Na maioria dos casos, os dados não correspondentes ao resultado esperado são um sintoma de uma causa ainda oculta. O processo de depuração tenta ligar o sintoma à causa, levando assim à correção do erro.

A idéia descrita acima parece muito simples e fácil de ser aplicada a um sistema, mas é sabido que o teste de software tem um custo muito alto e devido a isso há por trás várias atividades envolvidas. O plano de teste auxilia nessas atividades. A primeira parte de um plano de teste envolve determinar quais objetivos e qual estratégia de teste usar. O plano também deve abordar os recursos utilizados, os processos a serem seguidos, as técnicas e critérios a serem aplicados, o tempo disponível e outros aspectos. Como um plano de sistemas, o plano de teste não é um documento estático, ele sofre alterações juntamente com

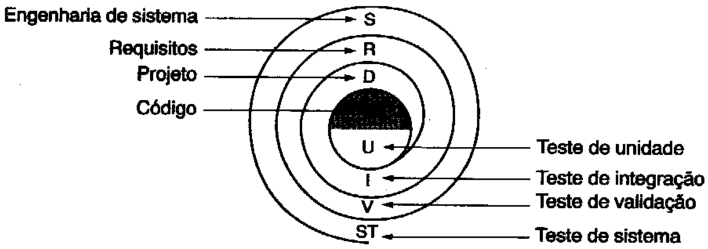

Figura 1: Uma estratégia de teste

o desenvolvimento do sistema e o andamento do teste. O planejamento de teste permite que a equipe técnica envolvida consiga uma visão geral do que realmente precisa ser testado no sistema[5, 18].

Um *processo de teste* consiste em estabelecer passos que correspondam às etapas do processo de engenharia de software(Engenharia de sistemas, análise de requisitos, projeto e implementação) [5]. Assim como o processo de software, o processo de teste também pode ser visto como um espiral[14], como mostra a figura 1. O processo de teste é dividido em quatro estágios:

1. *Teste de unidade*: analisa cada unidade ou componente a ser testado. Todos são testados individualmente para que se tenha a certeza que eles operam corretamente. Neste caso são analisadas as deficiências algorítmicas e as estruturas de dados. A estrutura de dados é examinada para garantir que os dados armazenados temporariamente mantém sua integridade durante todos os passos da execução de um algoritmo. As deficiências algorítmicas são examinadas para garantir que o sistema faça o que realmente é esperado que ele faça;

2. *Teste de integração*: As funcionalidades de cada unidade necessitam ser testadas, e as entidades funcionais maiores são construídas através destas unidades. A combinação dessas unidades é feita até que o programa, o subsistema ou o sistema seja montado. Este processo é chamado de integração do sistema ou do programa. As unidades combinadas necessitam também serem testadas para ver se funcionam corretamente quando unidas. Este tipo de teste é chamado de teste de integração. Neste caso, focaliza-se em testar as interfaces e as interdependências das unidades que foram combinadas. As diferentes estratégias de integração dos módulos diferem em como eles se aproximam e em que seqüência a combinação e a integração deles são feitas até que o sistema, ou o programa, seja construído[18]. Duas estratégias são bastante utilizadas nesse caso: *top-down* e *bottom-up*. Na

integração *top-down* os módulos são integrados movimentando-se de cima para baixo através da hierarquia de controle, iniciando do módulo de controle principal aos módulos subordinados. Já a integração *bottom-up* tem a característica inversa da *top-down*. A escolha de uma estratégia de integração depende das características do software, mas uma abordagem combinada de ambas estratégias pode ser o melhor ajuste;

3. *Teste de validação*: este estágio tem a preocupação com relação ao software construído, valida-se os requisitos de software estabelecidos na fase de análise. O teste de validação pode ser dito bem sucedido quando o software funciona de uma maneira razoavelmente esperada pelo cliente;

4. *Teste de sistema*: este estágio analisa o teste do software com outros elementos do sistemas, ou seja, por completamente à prova o sistema baseado em computador;

Esta seção relatou a importância de um planejamento e de um processo de teste. O processo de teste foi descrito de uma maneira simples. Na próxima seção, serão exploradas as técnicas de teste mais utilizadas para se conduzir uma atividade de teste. Em cada técnica será feito um estudo das dificuldades das técnicas junto com os processos de teste.

1.4 Técnicas de teste

Em cada um dos estágio do processo descrito acima, são selecionados sub-conjuntos de dados de teste a serem utilizados. Mas, por que selecionar sub-conjuntos de dados de teste e não utilizar um conjunto de dados de teste como um todo? Realizar o teste em todas as combinações possíveis de valores de entrada para um programa, não é viável. Provar a corretude de um programa inteiro é considerado inviável, pois o número de combinações é muito grande. O ideal seria testar as partes consideradas críticas do sistema. Cada estágio do processo de teste é realizado mediante uma série de *técnicas sistemáticas* que auxiliam o projeto de casos de teste. As diversas técnicas de testes utilizam diferentes tipos de informação visando escolher bons dados de teste que aumentem a probabilidade de novos defeitos. Junto às técnicas, os *critérios de teste* têm auxiliado esse objetivo. Os critérios estabelecem requisitos a serem satisfeitos, orientando na seleção de dados de teste, na avaliação da qualidade do teste e na definição de requisitos de suficiência a serem atingidos[5].

Nas próximas subseções são descritas duas técnicas utilizadas: *teste estrutural* e *teste funcional*. Em geral, nenhuma delas é suficiente para garantir a qualidade da atividade de teste, devendo ser aplicadas em conjunto para se tentar assegurar um teste de qualidade[4, 5].

1.4.1 Teste Funcional ou Teste Caixa-Preta

O teste caixa-preta precisa saber o que o sistema deve fazer[13], pois ele está preocupado em examinar o comportamento das entradas e das saídas relacionas, desconsiderando a implementação do software. O teste caixa-preta também é conhecido como teste funcional, pois baseia-se na identificação de requisitos funcionais, ou seja, funções matemáticas são especificadas usando apenas suas entradas e saídas, sem o conhecimento da estrutura do código[18].

O teste funcional tem como objetivo identificar os seguintes erros [14]: funções incorretas ou ausentes; erros de interface; erros nas estruturas de dados ou no acesso a bancos de dados externos; erros de desempenho; e erros de inicialização e término.

A grande dificuldade de encontrar defeitos é selecionar as entradas que façam parte de um domínio de entradas. O *particionamento equivalente*[14] divide o domínio de entrada em classes de dados a partir das quais os casos de teste podem ser derivados. Os dados de entradas são separados em classes diferentes[2]. O nome particionamento equivalente é devido ao comportamento do programa, ou seja, todos os membros da uma classe têm comportamentos equivalentes no programa. Uma aproximação sistemática de falhas de teste é baseada na identificação de um conjunto de particionamento equivalente que deve ser tratado

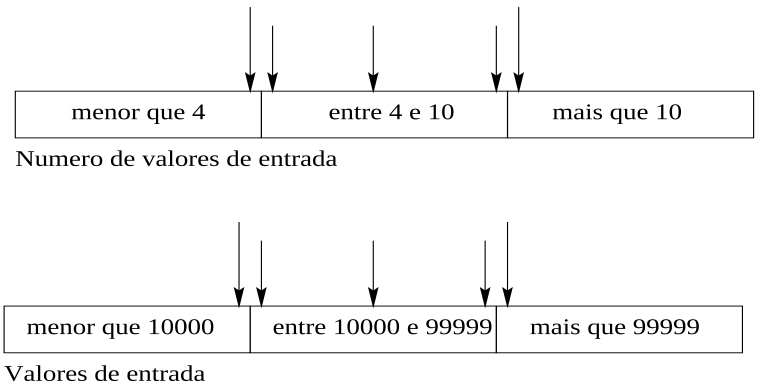

Figura 2: Exemplo de particionamento equivalente

por um programa. Os casos de teste são projetados de modo que entradas e saídas se encontrem dentro destas partições. Veja um simples exemplo onde a especificação do programa indica que o programa aceita de quatro a oito entradas e que cada entrada são de cinco dígitos inteiros maior que 10.000. A Figura 2 mostra os particionamentos equivalentes e os possíveis dados de entrada de teste. Para o teste

[2]números positivos, números negativos, strings sem espaço em branco e outros

ser completo, os dados de teste devem ser selecionados de modo que cada partição em cada classe seja testada com todos os exemplos das partições em outras classes [15].

Os métodos formais aplicados junto com *model-cheking*[3] estão auxiliando na derivação de casos de teste. O grupo de Gargantini[10] descreve um método para geração de casos de teste através de uma especificação de requisitos do SCR[4] (Software Cost Reduction) que contém tipos de variáveis misturados (inteiros, booleanos e tipos enumerados). O método para geração de conjunto de dados de teste cobre o conjunto de todos os possíveis dados de entrada organizando-os em classes de equivalência e gerando um ou mais casos de teste para cada classe de equivalência. Eles desenvolveram um ferramenta em Java que usa um model-checking para construir casos de teste de uma especificação de requisitos do SCR.

1.4.2 Teste Estrutural ou Teste Caixa-branca

O teste é baseado na estrutura do código. O uso da estrutura de fluxo de controle ajuda a guiar os testes usando um número de caminhos lógicos que serão testados, fornecendo casos de teste que põem à prova conjuntos específicos de condições e/ou laços[14, 18]. O código fonte é utilizado para identificar requisitos a serem cumpridos pelos casos de teste. A vantagem do teste estrutural é que uma análise do código pode ser usada para encontrar casos de teste que são precisos para garantir um dado *nível de cobertura atingido* no teste. O nível de cobertura está relacionado à utilização de critérios de teste como um meio para a avaliação da qualidade do teste. A análise dinâmica pode ser usada para medir a extensão desta cobertura e ajudar com projetos de casos de teste[15]. O nível de cobertura determina o percentual de elementos requeridos pelo critério de teste efetivamente exercitados pelo conjunto de dados de teste aplicados. Deste modo, quanto maior este valor, mais completo terá sido o teste[5].

Os casos de teste citados acima são gerados para garantir que todas as instruções do programa sejam exercitadas pelo menos uma vez durante os testes e que todas as condições lógicas também sejam exercitadas, segundo os critérios de teste estabelecidos. Os critérios de teste estrutural estabelecem componentes estruturais do programa a serem exercitados e satisfazer um desses critérios significa exercitar todos os componentes requeridos pelo mesmo[5]. Duas técnicas de teste estrutural bastante aplicadas são *teste de caminho básico* e *teste de fluxo de dados*.

O teste de caminho básico tem como objetivo auxiliar na geração de casos de teste que exercitem

[3]Model checking é uma técnica para verificação automática de sistemas: um algoritmo de model-checking determina se um modelo matemático de um sistema corresponde a especificação dos requisitos que é dado como uma fórmula de lógica temporal.[1]

[4]O método SCR foi desenvolvido em 1978 para especificar os requisitos do Operatinal Flight Program(OFP) do U. S. Navy's A-7 aircraft

todos os caminhos lógicos independentes[5], pelo menos uma vez, durante a atividade de teste[14, 15].A idéia é converter o código fonte em um grafo de fluxo, onde cada *nó* do grafo representa as decisões e os *arcos* representam o fluxo de controle. Ao ter o grafo formado, o projetista do caso de teste tem uma visão dos possíveis caminhos e com isso pode gerar casos de teste que exercitem esses caminhos pelo menos uma vez. Veja o exemplo abaixo de um simples programa:

while $(result > 0)$ **do** {1}
   pointer = false;{2}
   if $(X > K)$ **then** {3}
      pointer = true;{4}
   end if
   $X = X + 1$;{5}
   avalia(X,pointer,&result){6}
end while
print(result);{7}

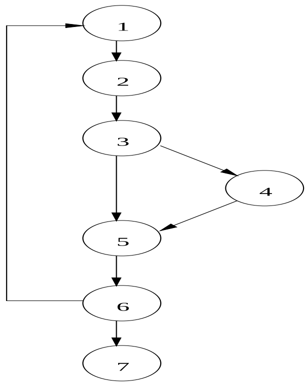

Figura 3: Grafo de Fluxo

Codificando esse programa para um grafo de fluxo, veja Figura 3, a idéia é gerar casos de teste que exercite todos os caminhos do grafo pelo menos uma vez. No caso desse simples programa, os caminhos são:

- 1-2-3-4-5-6-7
- 1-2-3-4-5-6-1
- 1-2-4-5-6-7
- 1-2-4-5-6-1

Um dos grande problemas do teste estrutural é a seleção desses caminhos. Existem dois tipos de caminhos executáveis e não executáveis. Um caminho é dito executável se existir um conjunto de valores

[5]Um caminho independente é qualquer caminho do programa que introduza pelo menos um novo conjunto de instruções de processamento ou uma nova condição[14]

para as variáveis de entrada, variáveis globais e parâmetros do programa que causam a execução do caminho. O caminho é dito não executável se tal conjunto não existir. A existência de caminhos não executáveis traz sérios problemas para o teste baseado em técnica estrutural, pois dado um critério C, nem sempre é possível gerar um conjunto de casos de teste T, tal que sejam exercitados todos os elementos requeridos por C e alcançar uma medida de cobertura igual a 100%. Um dos trabalhos para tentar tratar a não executabilidade foi proposto pelo grupo de Maldonato[17]. Eles Construíram uma ferramenta que auxilia a geração de dados de teste para elementos requeridos segundo um critério, com isso possibilita a eliminação automática de caminhos não executáveis, evitando um esforço e tempo gasto tentando cobrir tais caminhos e reduzindo os custos das atividades de teste.

Outra técnica de teste de estrutural é o teste de fluxo de dados que tem como objetivo verificar a lógica do programa[5, 14]. Esta técnica seleciona e executa caminhos de um programa do pondo onde uma variável foi definida, até o ponto onde ela foi utilizada. Veja o simples programa dado acima, as variáveis *pointer, X, K* e *result* todas são definidas no programa, a variável X além de ser definida também é usada. Veja na linha 6: $X=X+1$ a variável no lado esquerdo do comando de atribuição(=) está sendo definida, mas a direita do comando ela está sendo usada.

O teste estrutural permite que os desenvolvedores determinem se um programa é parcialmente consistente ou não com seu comportamento e com o projeto especificado através da análise de valores intermediários das variáveis durante a execução do programa. As saídas de um teste caixa-branca, geradas através do debugging ajuda a validar se o código está de acordo com o modelo do projeto. Após a execução do programa, os valores em um momento da execução são comparados aos valores que foram previstos. A inconsistência entre os valores previstos e os valores reais podem conduzir à descoberta de erros na especificação e na implementação. Alguns trabalhos usando teste de especificação junto com técnicas de *model checking* para verificar, organizar e gerar teste durante o desenvolvimento do software, auxiliam na descoberta de tais inconsistências. O desenvolvimento de técnicas para verificar parte da execução do sistema durante o teste estrutural usando um model checking tem sido um grande passo nessa área[6].

Um outro problema é a identificação da ocorrência de falhas na especificação. Isso é uma questão extremamente relevante, neste caso pressupõe a existência de um mecanismo para determinar se os valores de saída são corretos. Este mecanismo é conhecido como *oráculo*[6]. Como as especificações existem freqüentemente apenas em linguagem natural, a interpretação dos valores de saída é de caráter humano,

[6]Um oráculo de teste decide se ou não um resultado particular do programa está correto de acordo com a especificação.

com isso percebe-se que o oráculo não é automático. A existência de um oráculo é essencial para que o processo de teste faça sentido [5, 11]. Alguns métodos[8] para geração automática de oráculos de teste usando um model checking estão sendo desenvolvidas para resolver esse problema. Cada oráculo pode ser usado para forçar a execução do sistema num comportamento específico e verificar se o sistema se comporta dentro dos limites das partições esperadas. Se isso não ocorrer, houve um erro no modelo ou na implementação e cada falha deve ser tratada individualmente para determinar a origem da inconsistência. Os model checking incluem mecanismos para produzir contra-exemplos que sirvam como modelo de teste para que os oráculos possam verificar e direcionar um conjunto de teste numa implementação.

O teste é uma das partes mais caras do desenvolvimento do sistema e a aplicação de métodos formais pode auxiliar na redução desse custo. A combinação de análise de mutantes, model checking e geração de teste pode resolver alguns problemas plausíveis e produzir bons conjuntos de teste. A análise de mutantes é um método de teste estrutural para a geração de casos de teste que são sensitivas a alguma pequena mudança sintática na estrutura do programa. Uma idéia que está sendo muito utilizada é aplicar o model checking para processar mutantes e ao se encontrar uma inconsistência, o model checking gera um contra-exemplo que são convertidos automaticamente em casos de testes [12, 2].

1.5 Outros trabalhos relacionados

A atividade de teste é um elemento de um tema mais amplo que freqüentemente é chamado de *verificação* e *validação* (VI&V). A verificação analisa se o software está consistente e completo segundo sua especificação. A validação analisa se o software desenvolvido é consistente com as exigências do domínio e da aplicação, ou seja, se o software atende as expectativas esperadas pelo cliente[14, 15, 7]. Um trabalho desenvolvido por Easterbrook e Callahan[9] usa da especificação formal como uma ferramenta para a verificação e validação (VI&V) independentes. A intensão não é usar métodos formais como processo de desenvolvimento própria, mas como uma atividade paralela, executada por uma equipe independente. A idéia não é trabalhar em cima de toda a especificação mais sim com uma parte critíca da especificação. O processo de partes traduzidas de uma especificação em notação formal ajuda a detectar ambigüidades e aumentar a legibilidade, mesmo que a tradução seja apenas parcial. O processo também pode ser usado para impedir enganos, assim aumenta a confiança da equipe de IV&V de que a interpretação da especificação está correta. Os modelos formais parciais podem ser testados para encontrar as propriedades chaves do domínio e para validar as exigências originais. Tais propriedades são particularmente difíceis de serem analisadas com uma especificação informal.

A verificação concentra-se na demonstração da consistência entre uma implementação e uma especificação funcional. Para um sistema de segurança crítico isso não é o suficiente, a implementação também deve satisfazer as restrições de segurança do sistema e a segurança dos requisitos. O objetivo da verificação *safety* é determinar evidências para casos *safety*, mostrando que cada uma das circunstâncias de risco identificadas não podem ocorrer ou que a probabilidade de falha é pouco aceitável. Para atingir esse objetivo uma equipe da Universidade de York[16] vem desenvolvendo um framework para geração automática de dados de teste usando como suporte a análise estática. O resultado da análise estática é usado para dar critérios ao teste dinâmico. Estes critérios são usados para permitir a geração automática de dados de teste requeridos pelo teste dinâmico. Por outro lado, o teste dinâmico é usado para complementar e reduzir os custos envolvidos na atuação da análise.

O teste consome uma parcela significativa do orçamento para projetos do desenvolvimento do software. Os métodos formais, usados tipicamente nas fases da especificação e da análise do software, oferecem a oportunidade não somente de reduzir o custo do teste, mas também o aumento da confianca no software com a ajuda de métricas formais para melhorar o teste. A proposta de Ammann e Black[3] é aplicar o *model checking* e análise de mutantes ao problema de cobertura de dados de teste. O resultado da métrica de cobertura pode ser usada independentemente do código fonte, e é apropriada para o teste caixa-preta. Eles desenvolveram uma métrica baseada na análise de mutantes para avaliação dos dados de teste junto com o *model checking*.

2 Conclusão

A nossa proposta enfocou um estudo teórico de teste e verificação formal de sistema. Relatamos o objetivo de teste, o processo de teste e as duas técnicas de teste mais utilizadas, teste estrutural e teste funcional e junto a cada uma dessas técnicas relacionamos trabalhos que estão sendo aplicados e que enfatizam tais técnicas. Vimos alguns trabalhos relacinando teste e verficação formal em que a propriedade *safety* foi empregada. Esse trabalho contribuiu para aprimorar meu conhecimento na área de teste e verificação formal e com isso para que eu possa dar continuidade ao meu projeto de mestrado. Nosso próximo passo será estudar mais sobre verificação *safety* e também sobre semântica formal de java.

Referências

[1] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual Real-time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993.

[2] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specications. In *In Proc. 2nd IEEE Intern. Conf. on Formal Engineering Methods (ICFEM'98), Brisbane, Australia.*, pages 46–54. IEEE Computer Society, December 1998.

[3] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *In Proceeding of Fourth IEEE Ineternational High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society, November 1999. Also NIST IR 6403.

[4] Ellen Francine Barbosa, Auri Marcelo Rizzo Vincenzi, and José Carlos Maldonado. Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas C. In *XII Simpósio Brasileiro de Engenharia de Software*, pages 103–120, Outubro 1998.

[5] Paulo Marcos Siqueira Bueno. Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software. Master's thesis, Faculdade de Engenharia Elétrica e de Computação da Universidade de Campinas, 1999.

[6] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model checking. In *Proceedings 1996 SPIN Workshop*, August 1996. Also WVU Technical Report NASA-IVV-96-022.

[7] John R. Callahan and Steve M. Easterbrook. Verification and validation in a rapid software development process.

[8] John R. Callahan, Steve M. Easterbrook, and Todd L. Montgomery. Generating test oracles via model checking, 1998. Unpublished, working paper. Available at urlhttp://research.ivv.nasa.gov/projects/blurbs/FormalTesting.html.

[9] Steve Easterbrook and John Callahan. Formal methods for verification and validation of partial specifications: A case study. *The Journal of Systems and Software*, 40(3):199–210, March 1998.

[10] Angelo Gargantini and Constance L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Joint 7th European Software Eng. Conference and 7th ACM SIGSOFT Int. Symp. on Foundations of Software Eng.*, pages 146–162, September 1999. Toulouse, France.

[11] White Paper. Research issues in testing for software systems assurance.

[12] Paul E. Black Paul E. Ammann. Test generation and recognition with formal methods. The First International Workshop on Automated Program Analysis, Testing and Verification, June 2000.

[13] Shari Lawrence Pfleeger. *Software Engineering: the production of quality software.* Prentice-Hall, 1991.

[14] R. S. Pressman. *Software Engineering.* McGraw-Hill, 1997.

[15] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1996.

[16] Nigel Tracey, John Clark, John McDermid, and Keith Mander. Integrating safety analysis with automatic test-data generation for software safety verification. In *In the proceedings of 17th Internation System Safety Conference*, pages 128–137. System Safety Society, August 1999.

[17] Silvia R. Verfilio, José C. Maldonado, and Mario Jino. Caminhos não executáveis na automação das atividades de teste. In *VI Simpósio Brasileiro de Engenharia de Software*, pages 343–356, Novembro 1992.

[18] Anneliese von Mayrhauser. *Software Engineering: methods and management.* Academic Press, 1990.