**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**

Rita de Cássia Cazu Soldi

# AUTOMAÇÃO DE TESTE DE SOFTWARE PARA SISTEMAS EMBARCADOS

Florianópolis

2014

**SUMÁRIO**

# 1 INTRODUÇÃO

Sistemas embarcados podem ser apresentados como uma combinação entre *hardware* e *software* concebida para realizar uma tarefa específica. Estes sistemas estão amplamente acoplados a inúmeros dispositivos eletrônicos e suas atividades se tornaram muito populares e estão cada vez mais presentes no dia-a-dia das pessoas (CARRO; WAGNER, 2003). Controle de bordo automotivos, análise/monitoramento ambiental, eletrodomésticos, celulares e equipamentos de rede são apenas uma amostra de soluções proporcionadas através de sistemas embarcados.

Estas soluções baseam-se na premissa de que cada componente está efetuando corretamente as suas atividades e que a integração entre os mesmos não se desvia do comportamento esperado. Caso contrário, as consequências podem ser custosas como, por exemplo, falhas em sistemas financeiros, diminuição de uma determinada quota de mercado, perda/corrupção de informações importantes de clientes, etc (TASSEY, 2002).

Para garantir que as componentes vão agir de acordo com o requisitado, uma boa prática é testar de maneira pragmática cada detalhe do software. Sendo que pelo menos uma parte destes testes seja automatizada, de maneira que sirvam como um contrato de funcionamento do sistema.

Entretanto, o teste não é um comportamento do *software* e nunca deve interferir no fluxo de atividades que está sendo testado. Na maior parte dos sistemas isto é possível sem demandar muito esforço, mas sistemas embarcados costumam ser mais restritos em termos de: memória, capacidade processamento, tempo de bateria, prazos para executar uma determinada atividade, entre outros.

Além da dificuldade da própria atividade de teste, os desenvolvedores ainda precisam se adaptar a uma grande variedade de plataformas, sistemas operacionais, arquitetura, fornecedores, ferramenta de depuração, etc (SCHNEIDER; FRALEIGH, 2004).

Existem diversas ferramentas de apoio ao desenvolvimento de *software* embarcado que tentam minimizar o impacto destas variações. A escolha e integração destas ferramentas é uma etapa importante no construção de sistemas embarcados e deve ser feita de maneira criteriosa, uma vez que pode simplificar todo o processo de desenvolvimento.

## 1.1 MOTIVAÇÃO

Atualmente interagimos com mais de um sistema embarcado por dia e o número destes sistemas já superam o número de habitantes do nosso planeta, ainda, este número continua crescendo em ritmo acelerado (MARCONDES, 2009).

Além da quantidade, a complexidade dos sistemas embarcados é um fator que nunca para de crescer. O projeto de *hardware* e *software* está cada vez mais sofisticado e com requisitos mais rígidos. Para atender a esta demanda o *software* embarcado deixou de ser composto por um conjunto limitado de instuções *assembly* e deu espaço a uma gama de possibilidades oferecidas pelas linguagens de alto nível.

Mesmo com este acúmulo de atividades, o *software* embarcado ainda representa uma parcela minoritária do sistema, cerca de 20%, então grande parte das ferramentas de auxílio ao desenvolvimento ainda possuem poucas funcionalidades de validação e verificação. Ainda, constatou-se que as ferramentas disponíveis para automação de teste não são de fácil compreensão/execução e não possuem resposta satisfatória para os casos de falhas do caso de teste. Um ponto de melhoria seria pausar a execução no exato momento em que o *software* apresentar o comportamento inesperado e fornecer um relatório com os testes já realizados e oferecer dicas para que se possa detectar o *bug*.

Trabalhos recentes apontam que mais de 80% dos erros de um sistema embarcado provém do *software*, não do *hardware*, e que tanto o teste quanto a depuração são de suma importância para a qualidade do projeto embarcado. Desta forma, a motivação deste trabalho está em diminuir a dificuldade para realizar a verificação de sistemas embarcados, para que seja possível aproveitar melhor as oportunidades que são oferecidas pela indústria e tecnologia.

## 1.2 OBJETIVO

### 1.2.1 Objetivo Geral

O principal objetivo deste trabalho é identificar o estado da arte do processo de teste de *software* de sistemas embarcados. A partir deste estudo será proposta uma ferramenta para execução automatizada de testes de *software*. Além disso, esta ferramenta deve integrar-se automaticamente com um sistema de depuração de maneira simples e produzir informações que auxiliem na manutenção da qualidade do *software*.

### 1.2.2 Objetivos Específicos

Para atender o objetivo geral, os seguintes objetivos específicos devem ser concluídos:

- Planejar e executar uma revisão sistemática dos trabalhos relacionados, formando uma base de conhecimento sólida da área de teste de sistemas embarcados.
- Especificação de uma arquitetura que realize a automação de testes de maneira simples, sem que seja necessário configurar cada teste a ser executado.
- Especificação de um ambiente capaz de integrar a execução automática dos testes e a depuração de um *software*.
- Avaliação qualitativa e quantitativa da arquitetura proposta
- Apresentar o presente trabalho em forma de artigo científico em conferências e periódicos, para que especialistas da área de sistemas embarcados possam corroborar os resultados obtidos e a contribuição científica.

### 1.2.3 Delimitações

O presente trabalho não tem como objetivo a geração de casos de teste para um determinado *software*, limitando-se apenas em executá-los de maneira automatizada. Sendo assim, qualquer erro presente nos testes recebidos pelo protótipo para execução automática serão considerados como erro do próprio *software*.

# 2 CONCEITOS BÁSICOS

## 2.1 TESTE DE SISTEMAS COMPUTACIONAIS

Teste de *software* é uma área de pesquisa relativamente recente e teve sua primeira conferência formal foi organizada em junho de 1972 (GELPERIN; HETZEL, 1988). Desde então há um esforço para definir melhor os conceitos relacionados à área e também para chegar a um consenso sobre como documentação de testes deve ser feita.

A definição com maior aceitação é a padronizada pela IEEE, na qual um o teste de *software* é o processo de análise de um item de *software* para detectar as diferenças entre as condições existentes e exigidas (ou seja, erros) e avaliar as características do item de software (ANSI/IEEE, 1986). É importante ressaltar que o teste de *software* visa ressaltar a presença dos erros e não sua ausência, ou seja, o sucesso do teste mostra que o *software* analisado possui um desvio do comportamento previsto, mas quando o teste não encontra erros (inconclusivo) não se sabe se há ou não desvio no comportamento do *software* analisado.

Testar um *software* é essencial para revelar o número máximo de erros a partir do menor de esforço e esta atividade deve começar logo no início do projeto, com a definição dos requisitos a serem testados, da abordagem utilizada, da política de testes, critérios para a conclusão do teste, entre outros. Para que o processo de teste alcance seu objetivo é necessária uma estratégia de testes, suficientemente flexível para modelar-se às peculiaridades do *software* e rígida o bastante para um acompanhamento satisfatório do projeto.

Segundo Pressman (PRESSMAN, 2011), muitas estratégias de testes já foram propostas na literatura e todas fornecem um modelo para o teste com as seguintes características genéricas:

- Para executar um teste eficaz, proceder as revisões ténicas eficazes. Fazendo isso, muitos erros serão eliminados antes do começo do teste.

- O teste começa no nível de componente e progride em direção à integração do sistema computacional como um todo.

- Diferentes técnicas de teste são apropriadas para diferentes abordagens de engenharia de *software* e em diferentes pontos no tempo.

- O teste é feito pelo desenvolvedor do *software* e (para grandes projetos) por um grupo independente de teste.

- O teste e a depuração são atividades diferentes, mas depuração deve ser associada com alguma estratégia de teste.

As características gerais enumeradas por Pressman se referem a uma abordagem estratégica para teste de *software*. Para sistemas embarcados o teste também tem como objetivo aumentar a confiabilidade do sistema, pois sistemas embarcados estão intríscincos no dia-a-dia das pessoas e um erro neste tipo de sistema pode ser fatal. Então é desejável que a estratégia de teste de um sistema embarcado englobe:

- Teste de baixo nível - para verificar se os requisitos de *software* de baixo nível estão sendo atendidos e apontar se um pequeno segmento de código fonte foi implementado corretamente.
- Teste de integração de *software* - para verificar se o funcionamento do *software* está de acordo com os requisitos e analisar o relacionamento/comportamento entre os próprios componentes do *software* e dos componentes em relação à arquitetura do *software*.
- Teste de integração *software*/*hardware* - para verificar se a execução do *software* no *hardware* alvo ocorre corretamente e conforme o especificado.

Sendo assim, espera-se que os testes do sistema embarcado seja mais completos e robustos, a fim de descobrir os erros antecipadamente e, desta forma, garantir maior segurança ao *software* final.

## 2.2 DEPURAÇÃO DE SISTEMAS COMPUTACIONAIS

A depuração é a arte de diagnosticar erros em sistemas e determinar a melhor forma de corrigi-los. Devido esta atividade partir de um sistema com algum tipo de problema, é muito comum que a depuração ocorra como consequência de um teste bem sucedido.

Como o erro pode ser encontrado por qualquer pessoa e em qualquer etapa do ciclo de vida do *software*, tanto o formato do relatório de um resultado inesperado quanto as características do próprio erro podem variar de acordo com o conhecimento do autor do relato. Sendo assim, para realizar uma depuração eficaz deve-se ser capaz de identificar a técnica apropriada para analisar diferentes tipos de relatórios e obter as informações necessárias para eliminar o problema.

A depuração é uma tarefa complexa, cuja dificuldade varia de acordo com o ambiente de desenvolvimento, linguagem de programação, tamanho do sistema e disponibilidade de ferramentas disponíveis para auxiliar o processo.

Depuradores são ferramentas que permitem monitorar a execução de um programa, como opções como executar o programa passo a passo, pausar/reiniciar a execução e, em alguns casos, até voltar no tempo e desfazer a execução de uma determinada instrução.

A conexão entre o programa e o depurador pode ocorrer localmente ou remotamente. Ambas possuem vantagens e pontos de melhoria que devem ser consideradas para construir um ambiente de depuração estável.

O método local é quando o programa é executado na mesma máquina que o depurador, de modo que o processo tem uma latência mais baixa e há grande influência entre ambos. Por exemplo, se um processo provoca um *crash*, o depurador só poderia formar a hipótese de *crash* porque ele mesmo realizou um comportamento inesperado (travar ou reiniciar).

Já na depuração remota não ocorre este tipo de interferência, uma vez que aplicação e execução do depurador encontram-se em máquinas separadas.Do ponto de vista do ambiente de depuração, a depuração remota é semelhante à uma depuração local com duas telas conectadas em um único sistema.

Além da interferência e da latência, também deve-se utilizar de ferramentas aliadas para delimitar outros importantes conceitos envolvidos na depuração, tais como, a forma de configurar o modo de execução do código, como observar as saídas da aplicação, como verificar uma determinada variável de ambiente, como visualizar o *log* das tarefas realizadas, entre outros.

## 2.3 PROJETO DE SISTEMAS ORIENTADO À APLICAÇÃO

A metodologia de projeto de sistemas orientado à aplicação (***AOSD - acrônimo para *Application-Oriented System Design*) tem como objetivo principal a produção de sistemas para aplicações de computação dedicada e adaptados para atender exigências específicas da aplicação que irá utilizá-lo (FRÖHLICH, 2001).

Esta metodologia permite manter o foco nas aplicações que utilizarão o sistema que desde o início do projeto, então tanto a arquitetura quanto os componentes podem ser definidos a fim de maximizar a reutilização e a configurabilidade do sistema de acordo com as peculiaridades da sua aplicação. Os principais conceitos envolvidos em sua utilização são:

**Familias de abstrações independentes de cenário**

Durante a decomposição de domínio as abstrações identificadas são agrupadas em familias de abstrações e modeladas como membros desta familia. As abstrações devem ser modeladas de forma totalmente independente de cenários, garantindo que sejam genéricas o suficiente para

que sejam reutilizadas para compor qualquer sistema. Dependências ambientais observadas durante a decomposição de domínio devem ser separadamente modeladas como aspectos de cenário.

**Características Configuráveis**
Utilizadas quando uma determinada característica pode ser aplicada a todos os membros de uma familia, mas com valores diferentes. Então ao invés de aumentar a familia modelando novos membros, esta característica compartilhada é modelada como uma configurável. Desta forma é possível definir o comportamento desejado e aplicá-lo às abstrações de forma semelhante aos aspectos de cenário.

**Adaptadores de cenário**
Utilizados para aplicar os aspectos de forma transparente às abstrações do sistema. Eles funcionam como uma membrana, que envolve uma determinada abstração e se torna um intermediário na comunicação dessa abstração, inserindo as adaptações necessárias para cada requisição que seja dependente de um cenário.

**Interfaces infladas**
Resumem as características de todos os membros da familia em um único componente de interface, permitindo que a implementação dos componentes de *software* considere sempre uma interface bem abrangente e, desta forma, postergando a decisão sobre qual membro da familia utilizar. Esta decisão pode ser automatizada através de ferramentas que identificam quais características da familia foram utilizadas e selecionam o membro dessa familia que implementa o subconjunto da interface inflada.

A Figura1 mostra uma visão geral da ***AOSD, metodologia na qual é possível visualizar a decomposição do domínio em famílias de abstrações independentes de cenários e das dependências ambientais - separadamente modeladas como aspectos de cenário. A arquitetura do sistema é capturada pelos *frameworks* dos componentes modelados a partir do domínio. Somente estes componentes serão reunidos para formar o sistema, pois somente eles são necessários para fornecer suporte à aplicação.

## 2.4 PROGRAMAÇÃO GENÉRICA

A programação genérica pode ser abordada de vários pontos de vista. Para entender os conceitos básicos utilizados neste presente trabalho será utilizada a visão de Musser e Stepanov:

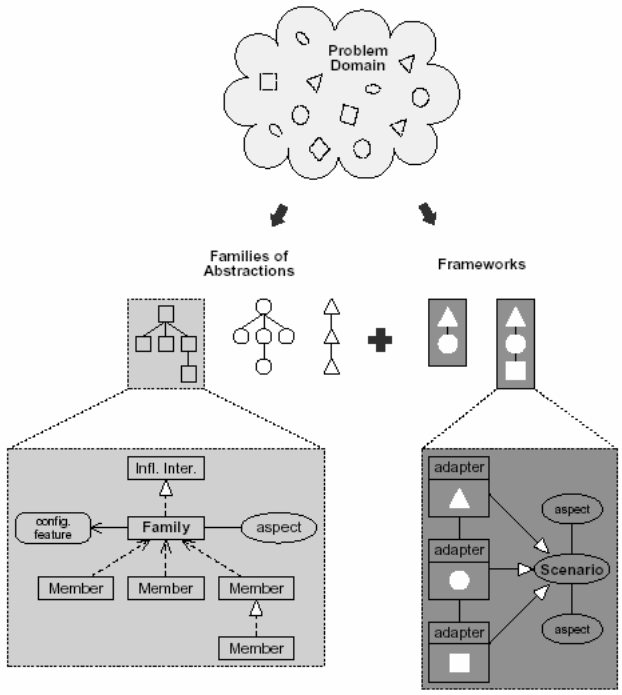


Figura 1 – Visão geral da metodologia de projeto de sistemas orientado à aplicação (FRÖHLICH, 2001)

> ”By generic programming, we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete efficient algorithms”(MUSSER; STEPANOV, 1989).

Esta noção de programação genérica foi a que motivou o projeto da ***STL e defende um paradigma de programação para a concepção e desenvolvimento de estruturas reutilizáveis e algoritmos eficientes.

Os algoritmos podem ser vistos como procedimentos parametrizados completamente independentes da representação de dados e são suficientemente genéricos para manter a mesma semântica e eficiência do algoritmo original, mesmo quando instanciado e assumindo tarefas simultâneas.

### 2.4.1 Utilização em C++: *templates*

Mecanismos de *templates* providos pelo C++ são uma maneira de adicionar à classes e funções conceitos genéricos, isto é, permitem criar um código reutilizável onde os tipos sejam passados como parâmetro.

No *template* ficam descritas classes/funções genéricas que podem variar sua declaração de acordo com um determinado argumento fornecido como parâmetro para o modelo.

É função do compilador gera as novas classes/funções quando houver os argumentos necessários para selecionar o parâmetro correto para instanciar o *template*. As classes/funções geradas a partir de um *template* são especializações deste modelo.

Um exemplo de implementação de classe genérica utilizando *templates* pode ser observado na Figura 2, onde letra *T* representa o parâmetro genérico que será trocado pelo compilador quando o *template* for instanciado.

```
template <class T> class Vector{
   T* v;
   int sz;
public:
   Vector();
   Vector(int);

   T& elem (int i) {return v[i]; }
   T& operator [] (int i);

   void swap(Vector &);

   // ...
};
```

Figura 2 – Classe *Vector* generalizada (STROUSTRUP, 1994)

Supondo que o programa apresente posteriormente a seguinte declaração:

```
Vector<int> vector_i;
Vector<MyClass> vector_my_class;
```

O Resultado seria a instanciação de duas classes diferentes de acordo com o argumento, como apresentado na Figura 3 e Figura 4:

Por padrão, as classes *template* possuem uma única implementação para qualquer argumento recebido e quando deseja-se dar um tratamento mais

```
class Vector{
   int* v;
   int sz;
public:
   Vector();
   Vector(int);

   int& elem (int i) {return v[i]; }
   int& operator [] (int i);

   void swap(Vector &);

   // ...
};
```

Figura 3 – Classe *Vector* instanciada com *int*.

refinado para atribuir uma determinada característica é necessário realizar descrições alternativas do *template*. Estas descrições, também conhecidas como especializações, são escolhidas pelo compilador com base nos argumentos fornecidos para o `template`.

```
class Vector{
   MyClass* v;
   int sz;
public:
   Vector();
   Vector(int);

   MyClass& elem (int i) {return v[i]; }
   MyClass& operator [] (int i);

   void swap(Vector &);

   // ...
};
```

Figura 4 – Classe *Vector* instanciada com *MyClass*.

# 3 TRABALHOS RELACIONADOS

A área de automação de testes possui uma vasta literatura de apoio, sendo a maioria relacionada a sistemas de propósito geral.

## 3.1 JUSTITIA

Justitia (SEO et al., 2007) é uma ferramenta de testes automatizados capaz de detectar automaticamente erros na interface, gerar casos de testes completos (entradas, estados internos, valores de variáveis, saídas) e emular a execução do *software* na arquitetura alvo.

Ela surgiu como um estudo de caso para demonstrar a automação de testes para *software* embarcado, em que os autores propuseram em esquema de teste automatizado para as interfaces das camadas do sistema embarcado baseado em sua emulação na arquitetura alvo.

Para defender que a interface é um critério essencial para um teste de *software* embarcado, a base de Justitia consiste na união de duas técnicas:

**Técnica de teste de interface** é reponsável por gerar casos de teste através da análise do arquivo (imagem) executável do sistema. Como o próprio nome sugere, a técnica foca nas interfaces do sistema embarcado, em especial as referentes às camadas do sistema operacional e às camadas do *hardware*.

**Técnica de emulação dos casos de teste** é uma combinação entre o monitoramenmento e a depuração do sistema. A ideia é definir *breakpoints* para os casos de teste da interface e monitorar a tabela de simbolos para definir o sucesso ou falha do teste.

Os resultados foram analisados através de experimentos sob três pontos de vista: a densidade da interface, a complexidade do *software* e a relação interface *versus* erros.

A primeira questão a ser investigada foi a da densidade da interface, pois que apesar de estar estritamente ligada ao grau de acoplamento do *software* embarcado, não há uma relação definida entre o grau de acoplamento *software* e sua testabilidade. A partir da primeira questão houve a necessidade de estabelecer uma outra relação entre a densidade e a complexidade de um *software*, que atualmente é uma das principais métricas para a previsão de falhas.

O primeiro protótipo apresentou resultados satisfatórios para as questões acima e chegou-se a conclusão de que a interface deve ser utilizada como um

*checkpoint* para encontrar e corrigir erros. A técnica foi novamente corroborada por uma nova versão de Justitia (KI et al., 2008), agora aplicada ao teste de um *driver* de um dispositivo embarcado e com a capacidade de suportar um número ainda maior de testes.

## 3.2 AUTOMATIC TESTING ENVIRONMENT FOR MULTI-CORE EMBEDDED SOFTWARE (ATEMES)

*ATEMES* (KOONG et al., 2012) é uma ferramenta para automação de teste para sistemas embarcados de multiplos núcleos. Dentre os tipos de teste suportados estão os aleatórios, de unidade, cobertura, desempenho e condições de corrida. A ferramenta também prevê instrumentação do código, geração de casos de uso e de dados de entrada para sistemas de múltiplos núcleos.

Para desempenhar todas estas funções *ATEMES* conta com os seguintes componentes:

**PRPM** é módulo que realiza o de pré-processamento. Dentre suas atribuições estão a análise de código fonte, geração automatica de casos de teste, geração automática dos dados de entrada de teste e instrumentação do código fonte. Suas funções são automáticas, mas também é possível intervir manualmente nos testes através de uma interface.

**HSATM** é uma das partes do módulo de teste automático, a que pertence ao lado anfitrião. Responsável por gerar automaticamente uma base para os testes baseados em uma biblioteca pré-definida, compilar o código fonte para uma determinada arquitetura de *hardware* e enviar a imagem executável para a plataforma alvo.

**TSATM** também compõe o módulo de teste automático, mas com a foco voltado à plataforma de *hardware* alvo. Sua principal função é executar a imagem recebida do *HSATM*, monitorar o andamento dos testes e enviar os dados desta execução.

**POPM** é módulo de pós-processamento, o qual analisa todos os resultados e dados coletados durante o teste.

A partir destes componentes é possivel executar os testes em uma plataforma alvo a partir de uma estação de trabalho remota, o que diminui a quantidade de restrições de recursos dos sistemas embarcados. Para tanto, o *software* é sofre uma compilação cruzada no lado anfitrião (estação de trabalho) e é enviado automaticamente para a plataforma alvo, onde é que o executado.

As atividades são gravadas em um *log*, dentre estas estão os dados de execução do sistema, o tempo de utilização de cada núcleo da CPU durante a execução dos testes, o resultado de saída, entre outros. O *log* pode ser passado para o lado anfitrião em tempo de execução, onde pode ser armazenado, processado e até apresentado de maneira visual através de uma interface.

## 3.3 STATISTICAL DEBUGGING

Trabalhos com *statistical debugging* se utilizam dados estatísticos relacionados a várias execuções do sistema para isolar um *bug*. Esta análise reduz o espaço de busca utilizando-se de recursos estatisticamente relacionados ao fracasso, limitando assim o conjunto de dados até chegar a uma seleção em que o erro se faz presente.

Apesar do espaço de busca minimizado, a técnica retorna alguns locais espalhados no código, o que não facilita o trabalho do desenvolvedor. Para tentar suprir a necessidade de mais informações, foram sugeridas as extrações de algumas métricas destes dados estatísticos. Um exemplo seria o *ranking* de desconfiança, que é uma relação entre uma determinada característica versus seu o número de execuções com falha, classificados em ordem decrescente.

Como a análise é realizada a partir de uma depuração estática, estes modelos expõem as relações entre comportamentos do *software* e seu eventual sucesso/fracasso de uma maneira estática. Então é possível fornecer uma identifiçao aproximada de qual parte do sistema gerou o erro.

Devido ao grande volume de dados necessários para guardar todas as execuções e realizar toda esta análise, esta técnica é difícil de ser implementada em um sistema embarcado real.

### 3.3.1 Statistical Debugging of Sampled Programs

Esta abordagem propõe recolher os dados estatísticos através de declarações inseridas no código e que podem coletar dicas, em tempo de execução, sobre os dados que não estão relacionados com as falhas (ZHENG et al., 2003).

Um dos desafios desta proposta é coletar os dados necessários sem penalizar a execução do *software* e garantindo a melhor utilização de todos os recursos do sistema. A solução encontrada foi inserir um evento aleatório junto às declarações inseridas no *software*, possibilitando que apenas uma pequena parte delas seja executada para cada nova execução do sistema.

A partir desta probalidade foi possível reduzir tempo gasto para coletar os dados, já que não é obrigatório executar todas as declarações em todas as

execuções do sistema. Além disso, as amostras recebidas são agregadas sem a informação de cronologia e considerando apenas a quantidade de vezes em que o resultado das declarações permaneceu o mesmo, o que minimiza o espaço necessário para armazenar os dados.

### 3.3.2 Statistical debugging: simultaneous identification of multiple bugs

Este modelo de *statistical debugging* tem o propósito de identificar a origem dos erros em um *software* com uma modelagem probabilística de predicados, considerando inclusive que o *software* pode conter mais de um erro simultaneamente (ZHENG et al., 2006).

Os modelos baseados em *statistical debugging* têm como características possuir dos dados esparsos como amostragem e uma grande variedade de predicados para manipular. Por isso a manipulação destes predicados nem sempre é um trabalho trivial, o que faz com que muitos modelos não consigam selecionar padrões de erros úteis. Para mitigar este problema foram utilizadas técnicas semelhantes às dos algoritmos *bi-clusters*, que permite a extração de dados bidirecional simultâneamente.

A implementação da extração de dados bidirecional executa um processo de votação coletiva iterativo, no qual todos os predicados tem um número que define a qualidade de representação, que pode ser modificado durante a execução do algoritmo através da distribuição de votos. Cada execução em que ocorre um erro tem direito a um voto para selecionar o predicado que melhor se encaixa na situação.

Esta solução é interessante porque reduz o problema de redundancia, pois o processo de votação só acaba quando hover convergência e quanto maior o número de predicados competindo pelo votos, menor é a quantidade de votos que cada um deles pode ter.

Depois de cada execução um predicado pode receber tanto um voto completo quando uma parcela do voto. No final, os predicados são classificados e selecionados considerando o número de votos que receberam.

### 3.3.3 Statistical debugging using a hierarchical model of correlated predicates

(PARSA; ASADI-AGHBOLAGHI; VAHIDI-ASL, 2011)

## 3.4 PROGRAM SLICING

Em técnicas que utilizam *program slicing* a ideia principal é a partição do código e a remoção de estados ou caminhos que são irrelevantes para alcançar o objetivo selecionado, o que no caso da verificação e validação de um *software* é encontrar um erro. Esta técnica possui duas abordagens principais: estática e dinâmica.

A partição estática é o tipo mais rápido e aponta apenas uma aproximação do conjunto final de caminhos que podem levar ao erro. Isto ocorre porque o foco é simplificar ao máximo o *software* em questão, reduzindo-o ao ponto de conter todos os estados que poderiam afetar o valor final, mas sem considerar o valor de entrada do *software*. Este tipo de partição é mais utilizada para *software* de pequeno porte e de pouca complexidade, em que o tamanho da partição permanece compatível com a sua simplicidade.

Já na partição dinâmica, as informações do critério de corte tradicional não são suficientes, sendo necessária uma informação adicional sobre os valores de entrada do *software*. A partição será realizada a partir destes dados e sequência de valores de entrada no qual o *software* foi executado é determinante para o conjunto de saída. Esta partição é mais utilizada para *software*s complexos e de alto grau de acoplamento.

Independente da abordagem selecionada, a partição do código é uma técnica interessante porque necessita apenas de uma execução com falhas para ser capaz de simplificar o grupo de entradas a serem examinadas.

### 3.4.1 (SASIREKHA; ROBERT; HEMALATHA, 2011)

### 3.4.2 (XU et al., 2005)

### 3.4.3 (ARTHO, 2011)

## 3.5 CAPTURE AND REPLAY

Em trabalhos que utilizam *capture and replay* a ideia é capturar toda a execução do programa até o final e armazenar as operações envolvidas em um *log*. Este tipo de sistema permite que o desenvolvedor controle a nova execução do *software* com execução de passos a frente, voltando passos na execução (contra o fluxo), examinando o contexto de alguma variável, verificando um determinado controle de fluxo, analisando fluxos alternativos, entre

outras possibilidades.

### 3.5.1 Pinpointing interrupts in embedded real-time systems using context checksums

Um grande desafio nesta técnica é como se adaptar a interrupções, pois toda a estrutura de reprodução do *software* se baseia no fluxo de controle e uma interrupção tem a capacidade de romper esta sequência. Uma interrupção pode ocorrer a qualquer momento e provocar uma ruptura no fluxo de controle, então a execução do programa para na instrução atual, e continua a rotina de tratamento de interrupção.

Neste aspecto, a solução de tirar um *snapshot* do contexto de execução quando ocorre uma interrupção (SUNDMARK; THANE, 2008) se mostra uma boa alternativa para que o desenvolvedor possa analisar erros oriundos de interrupções.

### 3.5.2 Replaying and isolating failing multi-object interactions

Burger e Zeller (BURGER; ZELLER, 2008) se destacaram por desenvolver uma ferramenta *JINSI* que consegue capturar e reproduzir as interações intercomponentes e intracomponentes. Assim, todas as operações relevantes são observadas e executadas passo a passo, considerando-se todas as comunicações entre dois componentes até encontrar o *bug*.

### 3.5.3 (QI et al., 2011)

### 3.5.4 (ORSO; KENNEDY, 2005)

# 4 O AMBIENTE COMPARTILHADO DE TESTE E DEPURAÇÃO

Este capítulo apresenta instruções de como integrar a execução de testes com os possíveis ambientes de depuração, gerando o ambiente compartilhado de desenvolvimento.

O ambiente utilizado pelo *script* de troca de parâmetros conta com um emulador para simular a execução da aplicação e rodar os testes. No caso de sucesso em um teste, automaticamente uma ferramenta de depuração é iniciada e o resultado de todas as operações é repassada para o usuário.

As ferramentas utilizadas neste ambiente podem ser substituídas por qualquer outra equivalente. A configuração será apresentada apenas para que o experimento possa ser reproduzido. Nele foi utilizado o QEMU para emular a máquina com a aplicação alvo a partir de outra máquina, usando tradução dinâmica. Desta forma torna-se possível utilizar um computador pessoal para testar aplicações compiladas para algum outro sistema embarcado.

Para dar suporte também à depuração, foi necessário procurar um aliado para ver quais os passos do programa foi executado um momento antes de um *crash*. O GBD - o GNU Project Debugger encaixou-se no papel de depurador, pois nele é possível especificar qualquer regra que possa afetar seu comportamento de maneira estática.

## 4.1 CONFIGURAÇÃO DO AMBIENTE REMOTO

A integração de ambos é particular para cada máquina anfitriã e alvo, portanto, talvez alguns passos aqui apresentados devam ser adaptados dependendo de sua arquitetura alvo. A Figura 5 apresenta as atividades necessárias para executar a depuração remota em conjunto com a simulação dos testes.

Uma explicação adicional das técnicas e ferramentas utilizadas neste processo estão listados abaixo:

**Compilar com informações de depuração** é o primeiro passo. Como entrada é necessário o código-fonte do aplicativo e como saída esperada encontra-se o código compilado com informações de depuração. Quem utilizar o gcc (GNU project C and C ++ compiler) pode realizar esta atividade simplesmente usando opção $-g$ para compilar.

**Emular o sistema utilizando um emulador** é um passo necessário para executar a aplicação na arquitetura alvo correta. Para executar este passo utilizamos o QEMU, qu deve ser inicializado com os argumentos $-s -S$. A primeira opção ativa o stub do para conectar um depurador, a fim

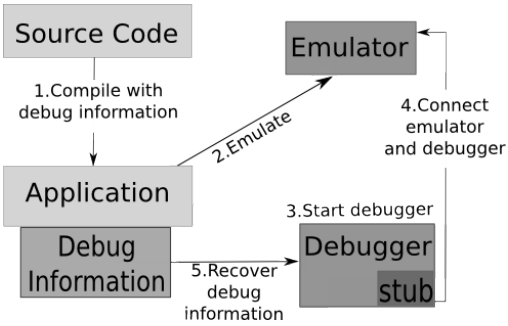


Figura 5 – Atividades de integração entre emulador e depurador

de abrir a comunicação entre o emulador e o depurador. A opção $-S$ é utilizada para forçar o QEMU esperar depurador para conectar antes da inicialização do sistema. Por exemplo, se uma aplicação compilada com informações de depuração (app.img) e imprime algo na tela (stdio), chamada QEMU deve ser semelhante à: $qemu - fdaapp.img - serialstdio - s - S$

**Conectar-se com o depurador** começa com uma sessão do depurador inicializada em uma janela separada. Então, para se conectar odepurador no QEMU o desenvolvedor deve explicitar que o alvo a ser examinado é remoto e informar o endereço da máquina alvo e porta em que se encontra o alvo a ser examinado (neste caso, o QEMU). Utilizando-se o GDB, a tela será iniciada a partir do comando: $targetremote[endereço\_alvo] : [porta\_alvo]$

**Recuperar as informações de debug** é um passo importante para ajudar os desenvolvedores a encontrar erros, pois este aquivo contém todas as informações que podem ser retiradas da compilação, como por exemplo o endereço de uma variável, ou os nomes contidos na tabela de símbolos. O arquivo usado para manter as informações de depuração deverá ser informado para o GDB usando o comando: $file[caminho\_do\_arquivo]$

**Encontrar origem dos erros** é uma atividade que depende do programa a ser depurado. A partir desta etapa, o desenvolvedor pode definir *breakpoints*, *watchpoints*, controlar a execução do programa e até mesmo permitir *logs*. Existem vários trabalhos com o foco em ajudar a encontrar os erros através da automação de alguns pontos, como a gerarão automática de *breakpoints* (ZHANG et al., 2013) e o controle do fluxo de execução (CHERN; VOLDER, 2007).

# 5 A ARQUITETURA DE AUTOMAÇÃO DE TESTES

A ideia de desenvolver a troca de parâmetros surgiu extrapolando-se conceitos da metodologia de projeto orientado aplicação (AOSD - *Application-Oriented System Design*) e do uso de programação genérica para a área de testes. O projeto orientado à aplicação fornece um sistema embarcado desenvolvido a partir de componentes especificamente adaptados e configurados de acordo com os requisitos da aplicação alvo. O fato de existir uma aplicação que fornece a certeza de que tudo que a compõe é essencial para seu funcionamento pode tornar o teste dos requisitos mais assertivo. Ainda, a programação genérica fornece uma maior adaptação do sistema às várias implementações de uma especificação.

No desenvolvimento para sistemas embarcados é frequente que uma única especificação seja reimplementada para atender a variabilidade de um componente de *software* ou *hardware*. Para cada uma destas implementações, um novo conjunto de testes é realizado. A vantagem de unir a AOSD com programação genérica ao desenvolvimento e teste de sistemas embarcados está em poder fazer uma única aplicação e um único teste para todas as implementações que seguem a mesma especificação, modificando apenas a configuração desejada.

O algoritmo de *TAP* (Troca Automática de Parâmetros de Software) é independente do sistema operacional e plataforma. No entanto, trabalhamos com a premissa de que este sistema seja orientado a aplicação, com modelagem baseada em *features* e parametrização. Também é desejável que cada abstração do sistema seja configurada conforme necessário através de *traits* de um modelo de *templates*, como o definido por Stroustrup (STROUSTRUP, 1994).

No algoritmo 1 são apresentados os passos a realizar a partir do momento em que se tem uma aplicação alvo até o retorno do relatório para o desenvolvedor. A entrada do algoritmo é o arquivo de configuração que possui o caminho da aplicação sob teste e de seus *traits*. A partir destas informações o algoritmo flui no sentido de tentar encontrar a característica desejada, trocá-la por uma valor predeterminado, executar a nova aplicação e recolher o retorno da aplicação.

A implementação atual utiliza o sistema operacional (`EPOS`) (FRÖHLICH, 2001), uma vez que adiciona uma grande capacidade de configuração do sistema, o que é muito adequado para avaliar o *script*. Para um melhor entendimento da implementação do algoritmo de `TAP` será necessária uma breve explicação de como configurar as abstrações no EPOS.

**Algoritmo 1:** Algoritmo de Troca dos Parâmetros de Configuração

```
   Entrada: arquivo # Arquivo de configuração do teste
   Saída: relatório # Relatório de tentativas
 1 propriedades ⇐ GetTraitFile(arquivo);
 2 se o arquivo possui valor de configuração então
 3     para cada configuração no arquivo faça
 4         linha ⇐ GetTheConfiguration(configuração,
           propriedades);
 5         para cada valor entre os da configuração faça
 6             novoPropriedade⇐ ExchangeValue(linha,
               propriedades) ;
 7             novaApp⇐ Compile(aplicação, novoPropriedade) ;
 8             relatório⇐ relatório + Emulate(novaApp) ;
 9         fim
10     fim
11 senão
12     se o arquivo possui n○ máximo de tentativas então
13         numMaxTentativas⇐ GetMaxSize(arquivo);
14     senão
15         numMaxTentativas⇐ GetRandomNumber();
16     fim
17     enquanto tentativas < numMaxTentativas faça
18         linha⇐ GetRandomNumber();
19         novoPropriedade⇐ ExchangeValue(linha, propriedades);
20         novaApp⇐ Compile(aplicação, novoPropriedade);
21         relatório⇐ relatório + Emulate(novaApp);
22     fim
23 fim
24 retorne relatório;
```

## 5.1 ABSTRAÇÕES NO *EPOS*

*EPOS* é um *framework* baseado em componentes que fornece todas as abstrações tradicionais de sistemas operacionais e serviços como: gerenciamento de memória, comunicação e gestão do tempo. Além disso, possui vários projetos industriais e pesquisas acadêmicas que o utilizam como base [1].

Este sistema operacional é instanciado apenas com o suporte básico para sua aplicação dedicada. É importante salientar que todas as características dos componentes também são características da aplicação, desta maneira, a escolha dos valores destas propriedades tem influência direta no comportamento final da aplicação. Neste contexto, a troca automatizada destes parâmetros pode ser utilizada tanto para a descoberta de um *bug* no programa quanto para melhorar o desempenho para a aplicação através da seleção de uma melhor configuração.

Cada aplicação tem seu próprio arquivo de configuração de abstrações para definir o seu comportamento. A Figura 6 mostra um trecho desta configuração para uma aplicação que simula um componente de estimativa de movimento para codificação H.264, o `DMEC`. Este trecho mostra como construir a aplicação, que neste caso foi configurada para executar no modo biblioteca para a arquitetura `IA-32` (*Intel Architecture, 32-bit*), através de um `PC` (*Personal Computer*).

```
template <> struct Traits<Build>
{
    static const unsigned int MODE = LIBRARY;
    static const unsigned int ARCH = IA32;
    static const unsigned int MACH = PC;
};
```

Figura 6 – Trecho do *trait* da aplicação o componente DMEC.

## 5.2 CONFIGURAÇÃO DO TESTE/DEPURAÇÃO

Para melhorar a usabilidade do *script*, é possível definir um arquivo de configuração com as informações necessárias para executar os testes unitários e de tipagem. Nós escolhemos XML para definir as configurações de teste,

[1] http://www.lisha.ufsc.br/pub/index.php?key=EPOS

pois pode definir todas as regras necessárias para executar o *script* de forma legível e, além disso, também é facilmente interpretado pelo computador.

A Figura 7 traz um exemplo do arquivo de configuração de TAP para a aplicação DMEC.

```
<test>
   <application name="dmec_app">
      <configuration>
         <trait>
            <id>ARCH</id>
            <value>IA32</value>
            <value>AVR8</value>
         </trait>

         <debug>
            <path>"/home/breakpoints.txt"<path>
         </debug>
      </configuration>
   </application>
<test>
```

Figura 7 – Exemplo do arquivo de configuração do teste para a TAP.

O arquivo do configuração é responsável pelo teste, então seu conteúdo deve estar sempre atualizado e em concordância com os requisitos da aplicação. O ajuste inicial é manual e simples, uma vez que este arquivo pode ser lido quase como um texto: há um teste para a aplicação (*philosopher_dinner_app*), dentro dela deseja-se especificar duas propriedades. A primeira é a propriedade identificada como *ARCH* que pode assumir os valores *IA32* ou *AVR8*. A segunda está relacionada à depuração, é um arquivo que contém *breakpoints* que está no seguinte caminho: "*/home/breakpoints.txt*".

## 5.3 DISCUSSÃO SOBRE A GRANULARIDADE NA CONFIGURAÇÃO DOS TESTES

Cada configuração do teste interfere diretamente com o tempo e eficácia do *script*. Prevendo este comportamento, TAP oferece três granularidades de configuração para o teste: determinada, parcialmente aleatória e aleatória. Elas devem ser escolhidas de acordo com a finalidade do usuário ao executar o *script* de troca de parâmetros, do tipo de teste e das características de aplicação.

Quando se deseja testar uma especificação bem definida, é possível de-

terminar qual valor uma propriedade deve atingir. Toda a especificação pode ser traduzida no arquivo de configuração e TAP só considerará sucesso no teste as execuções que seguirem fielmente o descrito. O modo determinado também é interessante quando se deseja otimizar uma configuração, pois uma vez que o comportamento da aplicação e todas suas configurações sejam conhecidas, a única variável do sistema afetará o resultado final.

Testes parcialmente aleatórios são usados para verificar as configurações do sistema que não possuem um valor determinado, ou seja, mais de um valor pode ser considerado correto. Neste caso, a informação faltante na configuração será atribuída pelo *script* no momento do teste. Sem nenhuma informação prévia, o algoritmo não garante que os valores gerados serão válidos e distintos uns dos outros, desta forma pode ser que o teste seja repetido e gere resultados com falsos negativos.

Teste aleatório foi desenvolvido como o pior caso. Ele só deve ser usado quando deseja-se testar valores fora do convencional para verificar a robustez da aplicação. Também é útil caso a aplicação falhe ao passar nos testes e não se tenha dica alguma sobre onde poderia estar o erro no momento de iniciar a depuração. Através dele pode-se encontrar valores errados de configuração e ajudar os desenvolvedores menos experientes a depurar pequenas aplicações.

# 6 RESULTADO EXPERIMENTAL DE TESTE E DEPURAÇÃO DE UMA APLICAÇÃO REAL

A troca automática de parâmetros foi utilizada para testar e depurar o componente de estimativa de movimento para codificação H.264, o *Distributed Motion Estimation Component* (*DMEC*). Este componente executa uma estimativa de movimento explorando a semelhança entre imagens adjacentes numa sequência de vídeo que permite que as imagens sejam codificadas diferencialmente, aumentando a taxa de compressão da sequência de *bits* gerada. Estimativa de movimento é uma fase importante para codificação H.264 já que consome cerca de 90% do tempo total do processo de codificação (LUDWICH; FROHLICH, 2011).

Teste de DMEC verifica o desempenho de estimativa de movimento usando uma estratégia de particionamento de dados, enquanto os trabalhadores *Workers*) realizam a estimativa e o coordenador (*Coordinator*) processa os resultados.

A Figura 8 apresenta a interação entre as *threads* coordenador e trabalhadoras. O coordenador é responsável por definir o particionamento de imagem, fornecer a imagem a ser processada e retornar resultados gerados para o codificador, enquanto cada trabalhador deve calcular o custo de movimento e os vetores de movimento.

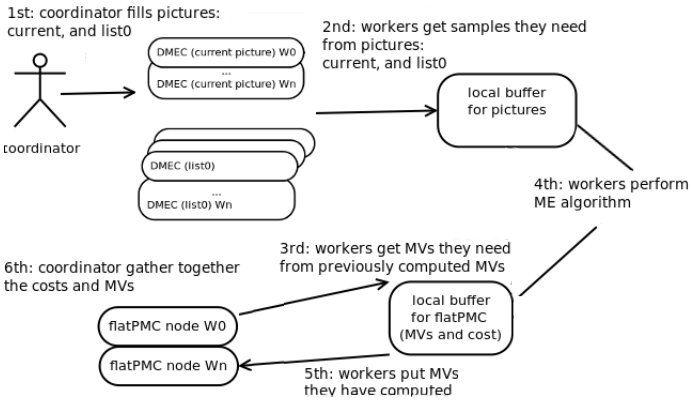


Figura 8 – Interação entre o coordenador e os trabalhadores na aplicação teste do DMEC (LUDWICH; FROHLICH, 2011).

Um dos requisitos do projeto era produzir as estimativas consumindo o menor tempo possível. Para tanto, houve a tentativa de aumentar o número de

trabalhadores para tentar paralelizar o trabalho da estimativa. A configuração *NUM_WORKERS* foi então testada para números entre 6 e 60. O teste do limite inferior e superior são demonstrados, respectivamente, na Figura 9 e na Figura 10.

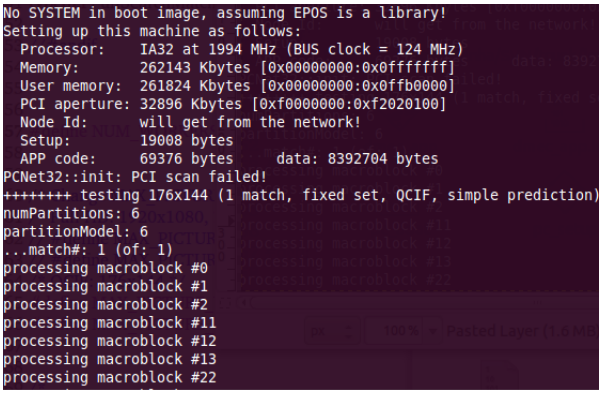


Figura 9 – Teste do DMEC com configuração NUM_WORKERS = 6

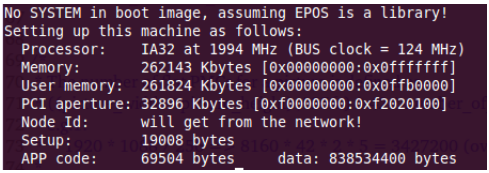


Figura 10 – Teste do DMEC com configuração NUM_WORKERS = 60

Apesar do *script* de troca de parâmetros gerar várias configurações para o teste, apenas compilar o código não garante que a aplicação é livre de *bugs*. No caso do número de trabalhadores igual a 60, o programa foi compilado, mas não foi possível emular sua execução. Nestes casos o depurador é automaticamente chamado para que se possa descobrir o porquê deste comportamento.

O *script* foi configurado para adicionar pontos de interrupção depois de iniciar cada uma das 5 funções da aplicação, inclusive na função principal, para descobrir se o problema encontrado era resultado de alguma delas. Foram consideradas corretas as execuções que contivessem então a resposta (*continue*) de cada uma delas. A Figura 11 mostra que não havia nenhuma resposta para a aplicação, informando que nem mesmo a função principal era atingida.

```
(gdb) target remote :1234
Remote debugging using :1234
0x0000fff0 in ?? ()
(gdb) file app/dmec_app
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from /home/tinha/SVN/trunk/app/dmec_app...done.
(gdb) b main
Breakpoint 1 at 0x8274: file dmec_app.cc, line 47.
(gdb) b testPack
testPack10()  testPack20()
(gdb) b testPack10()
Breakpoint 2 at 0x82f1: file dmec_app.cc, line 66.
(gdb) b testPack20()
Breakpoint 3 at 0x8315: file dmec app.cc, line 73.
(gdb) continue
Continuing.
```

Figura 11 – DMEC debug with GDB execution with NUM_WORKERS = 60

Observando o relatório final do *script* foi possível descobrir que sempre que a configuração NUM_WORKERS apresentava um número maior que 10 a aplicação se comportava de maneira anômala. Neste caso o conjunto de testes, depuração e relatório foi crucial para determinar o limite máximo de trabalhadores da aplicação.

## 6.1 RESULTADOS

Os testes foram realizados com a aplicação DMEC, executando sob EPOS 1.1 e compilado com GNU 4.5.2 e cross-compilados através de um computador pessoal com a arquitetura IA32. O ambiente integrado é composto por GDB 7.2 e QEMU 0.14.0.

A qualidade da informação de retorno é inerente à qualidade de informação de configuração do *script* de TAP. A Figura 12 apresenta um trecho de relatório com algumas configurações geradas.

Em casos como o do teste completamente aleatório qualquer propriedade pode mudar, por exemplo, o tamanho da pilha de aplicativos, o valor de um *quantum*, a quantidade de ciclos de relógio, etc. Estes relatórios são normalmente repetitivos e possuem informações espalhadas. Já nos relatórios gerados com mais dados tendem a ser mais organizados e repetirem menos informações.

As Figuras 13 e 14 apresentam, respectivamente, os resultados dos experimentos relacionados à qualidade da informação devolvida para o usuário e ao consumo de tempo. Neste experimento foram realizadas 50 tentativas para cada tipo de granularidade. Para o teste parcialmente aleatório, foi modificada a propriedade NUM_WORKERS com valores em aberto e para o teste determinado foi alterada esta mesma propriedade com valores de 1 a 60.

```
.*.*.*.*.* Test Report .*.*.*.*.*
Application= dmec_app

Original line = #define NUM_WORKERS 6
VALUES = 67,53,87,3,64,35,16,75,82,47,
79,70,81,12,46,84,68,18,76,26,
86,66,90,89,67,9,87,19,81,24,
31,2,12,24,58,33,15,3,55,4,
0,17,67,96,0,34,5,70,34,35,
27,41,40,88,94,45,96,7,55,72,
98,42,91,97,4,70,28,35,69,29,
34,19,28,72,15,96,29,39,87,72,
27,15,23,10,92,72,8,12,17,40,
62,42,17,90,45,83,35,81,10,7
```

Figura 12 – Trecho do relatório com a troca da propriedade `NUM_WORKERS` por valores gerados aleatoriamente.

A diferença entre as tentativas totalmente aleatórias e as outras duas granularidades foi grande. Este resultado já era esperado, visto que a depuração de uma aplicação sem informação nenhuma à priori tem a sua efetividade ligada à probabilidade de encontrar tanto a falha quanto a sua causa.

Entretanto não houve muita alteração entre os tipos determinado e parcialmente aleatório. Isto ocorreu devido à limitação na quantidade de propriedades e de seus possíveis valores de troca da aplicação, ou seja, com tal restrição as trocas com sucesso foram semelhantes nas duas configurações.

Conforme apresentado na Figura 15, a aplicação não tem uma imagem grande, mas quando adicionamos a informação extra em tempo de compilação, o consumo de memória foi aumentado em cerca de 200%. Em um sistema embarcado real, o tamanho desta nova imagem seria proibitivo.

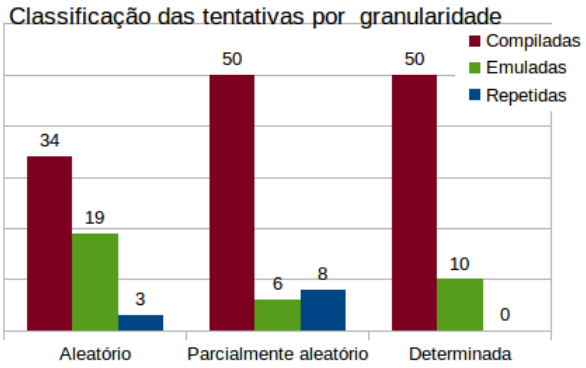


Figura 13 – Classificação das tentativas realizadas versus a configuração da granularidade.

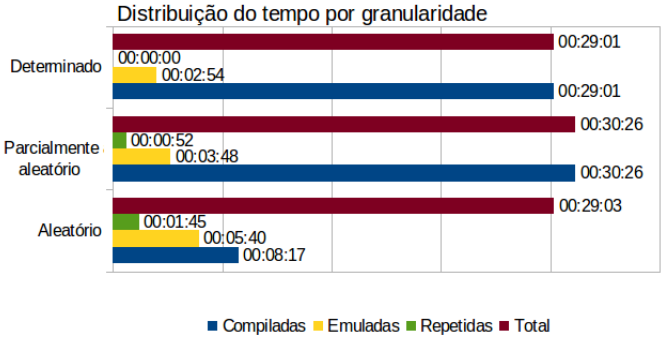


Figura 14 – Classificação das tentativas realizadas versus o consumo de tempo.

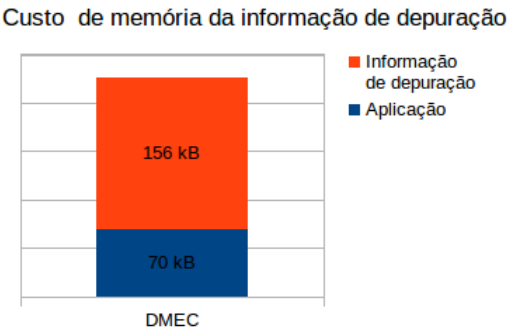


Figura 15 – Consumo de memória extra para armazenar as informações de depuração.

# 7 ANÁLISE QUALITATIVA ENTRE AS FERRAMENTAS DE TESTE E DEPURAÇÃO DE *SOFTWARE*

No Capítulo 3 foram apresentados vários estudos focados em apresentar soluções práticas para os diversos pontos de melhoria na área de teste e depuração de sistemas. Os mesmos trabalhos serão agora o foco de um estudo qualitativo de suas características.

A definição das características a serem analisadas foram insperadas na pesquisa de Antonia Bertolino(BERTOLINO, 2007), no qual são explicitados os conceitos mais relavantes para a área de testes de *software*, separados em realizações passadas e em metas ainda não atingidas. As metas para a pesquisa em teste de *software* são compostas por desafios relevantes a serem abordados, essenciais para o avanço do estado da arte. São eles:

1. **Teoria de teste universal** significa ter um padrão coerente de modelos/técnicas de teste, tornando possível averiguar os pontos fortes e as limitações de cada um deles e escolher racionalmente a melhor opção para cada caso.

   **Desafio: hipóteses explícitas.** Com a exceção de algumas abordagens formais, normalmente os testes são baseaddos em aproximações de uma amostra de dados inicial, suprimindo as demais informações das hipóteses. Entretanto, é de suma importância tornar estas informações explícitas, uma vez que esses pressupostos podem eluciar o porquê de observamos algumas execuções.

   **Desafio: eficácia do teste.** Para estabelecer uma teoria útil para testes, é preciso avaliar a eficácia dos critérios de teste existentes e novas técnicas que estão surgindo. Ainda não foi possível contextualizar e comparar a complexidade do teste do mundo real vesus o teste em ambiente controlado e nem refinar hipóteses nas bases para tais comparações. Por exemplo, mesmo a controvérsia convencional entre a técnica proposta versus técnicas aleatórias é amplamente utilizada até pelos métodos mais sofisticados. Este desafio aborda o porquê, como e quantos testes são necessários para descorir um determinado tipo de erro.

   **Desafio: testes de composição.** Este desafio está relacionado à como testar sistemas complexos. Tradicionalmente, a complexidade de teste tem sido abordada pela decomposição do teste em ensaios que testam separadamente alguns aspectos do sistema. Entretanto, ainda é preciso descobrir se a composição destes ensaios

é equivalente ao teste do sistema todo, entender como podemos reutilizar os resultados da decomposição e quais conclusões podem ser inferidas para o sistema a partir destes resultados.

**Desafio: evidências empíricas.** Na pesquisa de teste de *software*, estudos empíricos são essenciais para avaliar as técnicas e práticas propostas, para entender como e quando elas funcionam, e aperfeiçoá-las. Infelizmente, mais da metade do conhecimento existente é baseada em impressões/percepções dos autores, desprovida de qualquer fundamento formal. Para contribuir com o estado da arte de forma concreta é necessário realizar experimentos mais robustos e significativos em termos de escala, do contexto e do tema abordado.

2. **Modelagem baseada em teste**

   **Desafio: hipóteses explícitas de teste**

   **Desafio: eficácia do teste**

   **Desafio: testes de composição**

   **Desafio: Corpo de evidências empíricas**

3. **Testes 100% automaticos**

   **Desafio: hipóteses explícitas de teste**

   **Desafio: eficácia do teste**

   **Desafio: testes de composição**

   **Desafio: Corpo de evidências empíricas**

4. **Maior eficácia na engenharia de testes**

   **Desafio: hipóteses explícitas de teste**

   **Desafio: eficácia do teste**

   **Desafio: testes de composição**

   **Desafio: Corpo de evidências empíricas**

## 7.1 DEFINIÇÃO DAS CARACTERÍSTICAS

## 7.2 RESULTADOS

### 7.2.1 Justitia

### 7.2.2 ATEMES

### 7.2.3 Statistical Debugging

### 7.2.4 Program Slicing

### 7.2.5 Capture and Replay

# REFERÊNCIAS BIBLIOGRÁFICAS

ANSI/IEEE. *Std 1008-1987: IEEE Standard for Software Unit Testing*. [S.l.], 1986.

ARTHO, C. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer (STTT)*, Springer, v. 13, n. 3, p. 223–246, 2011.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: *Proceedings of the Future of Software Engineering at ICSE 2007*. [S.l.]: IEEE-CS Press, 2007. p. 85–103.

BURGER, M.; ZELLER, A. Replaying and isolating failing multi-object interactions. In: ACM. *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ISSTA 2008*. [S.l.], 2008. p. 71–77.

CARRO, L.; WAGNER, F. R. Sistemas computacionais embarcados. *Jornadas de atualização em informática. Campinas: UNICAMP*, 2003.

CHERN, R.; VOLDER, K. D. Debugging with control-flow breakpoints. In: *Proceedings of the 6th International Conference on Aspect-oriented Software Development*. New York, NY, USA: ACM, 2007. (AOSD '07), p. 96–106. ISBN 1-59593-615-7. <http://doi.acm.org/10.1145/1218563.1218575>.

FRÖHLICH, A. A. *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. (GMD Research Series, 17).

GELPERIN, D.; HETZEL, B. The growth of software testing. *Commun. ACM*, ACM, New York, NY, USA, v. 31, n. 6, p. 687–695, jun. 1988. ISSN 0001-0782. <http://doi.acm.org/10.1145/62959.62965>.

KI, Y. et al. Tool support for new test criteria on embedded systems: Justitia. In: KIM, W.; CHOI, H.-J. (Ed.). *Proceedings of the 2nd International Conference on Ubiquitous Information Management and Communication, ICUIMC 2008, Suwon, Korea, January 31 - February 01, 2008*. [S.l.]: ACM, 2008. p. 365–369. ISBN 978-1-59593-993-7.

KOONG, C.-S. et al. Automatic testing environment for multi-core embedded software (atemes). *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 85, n. 1, p. 43–60, jan. 2012. ISSN 0164-1212. <http://dx.doi.org/10.1016/j.jss.2011.08.030>.

LUDWICH, M.; FROHLICH, A. Interfacing hardware devices to embedded java. In: *Computing System Engineering (SBESC), 2011 Brazilian Symposium on*. [S.l.: s.n.], 2011. p. 176 –181.

MARCONDES, H. *Uma Arquitetura de Componentes Híbridos de Hardware e Software para Sistemas Embarcados*. 92 p. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2009.

MUSSER, D. R.; STEPANOV, A. A. Generic programming. In: *Symbolic and Algebraic Computation*. [S.l.]: Springer, 1989. p. 13–25.

ORSO, A.; KENNEDY, B. Selective capture and replay of program executions. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 2005. v. 30, n. 4, p. 1–7.

PARSA, S.; ASADI-AGHBOLAGHI, M.; VAHIDI-ASL, M. Statistical debugging using a hierarchical model of correlated predicates. *Artificial Intelligence and Computational Intelligence*, Springer, p. 251–256, 2011.

PRESSMAN, R. S. *Engenharia de software : uma abordagem profissional*. Sétima edição. [S.l.]: AMGH, 2011. ISBN 9788563308337.

QI, D. et al. Locating failure-inducing environment changes. In: ACM. *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. [S.l.], 2011. p. 29–36.

SASIREKHA, N.; ROBERT, A.; HEMALATHA, D. Program slicing techniques and its applications. *Arxiv preprint arXiv:1108.1352*, 2011.

SCHNEIDER, S.; FRALEIGH, L. The ten secrets of embedded debugging. *Embedded Systems Programming*, MILLER FREEMAN INC., v. 17, p. 21–32, 2004.

SEO, J. et al. Automating embedded software testing on an emulated target board. In: ZHU, H.; WONG, W. E.; PARADKAR, A. M. (Ed.). *AST*. [S.l.]: IEEE, 2007. p. 44–50. ISBN 0-7695-2892-9.

STROUSTRUP, B. *The design and evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994. ISBN 0-201-54330-3.

SUNDMARK, D.; THANE, H. Pinpointing interrupts in embedded real-time systems using context checksums. In: *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*. [S.l.: s.n.], 2008. p. 774–781.

TASSEY, G. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, Citeseer, p. 02–3, 2002.

XU, B. et al. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 2, p. 1–36, mar. 2005. ISSN 0163-5948. <http://doi.acm.org/10.1145/1050849.1050865>.

ZHANG, C. et al. Automated breakpoint generation for debugging. *Journal of Software*, v. 8, n. 3, 2013. <https://66.147.242.186/ academz3/ojs/index.php/jsw/article/view/jsw0803603616>.

ZHENG, A. et al. Statistical debugging of sampled programs. *Advances in Neural Information Processing Systems*, v. 16, 2003.

ZHENG, A. et al. Statistical debugging: simultaneous identification of multiple bugs. In: ACM. *Proceedings of the 23rd international conference on Machine learning*. [S.l.], 2006. p. 1105–1112.