4 AMBIENTE COMPARTILHADO DE TESTE E DEPURAÇÃO

Este capítulo apresenta o ambiente compartilhado de teste e depuração, demostrando de onde as ideias foram extraídas, adaptadas e, em alguns casos, melhoradas. Além disso, serão apresentas instruções para integrar a execução de testes com os possíveis ambientes de depuração, gerando o ambiente compartilhado de desenvolvimento.

A atividade de teste é necessária para manter a qualidade do produto oferecido e verificar se seu comportamento não desvia do esperado. No caso de *software* embarcado, o teste na plataforma alvo e no ambiente de produção é necessário, entretanto nem sempre é possível ou praticável. Felizmente existem alternativas para realizar esta atividade, como por exemplo executar o *software* em um simulador, através um protótipo limitado de *hardware*, ou até utilizando o próprio ambiente de desenvolvimento. (JGRENNING, 2004).

A ideia de prover um bom ambiente em que testes e depuração possam compartilhar a infraestrutura se formou a partir da observação e estudo de vários ambientes de execução automática das atividades testes e de depuração. Seu desenvolvimento envolveu uma atenção especial ao suporte para sistemas embarcados, visto que os testes/depuração realizados tentem a compartilhar os escassos recursos do SUT. O compartilhamento é crítico no caso de sistemas embarcados porque geralmente possuem apenas os recursos essenciais para desempenhar uma determinada função.

Um exemplo de sistema embarcado com um alto grau de dificuldade para atividades de teste e depuração é a Redes de Sensores Sem Fio (RSSF). Este tipo de rede pode ser composta por vários sensores que, além de permanecerem espalhados geograficamente por uma grande área de monitoramento, geralmente possuem pouco poder de processamento e apenas alguns *kilobytes* de armazenamento (POTTIE; KAISER, 2000).

No caso de muitas RSSF é impraticável coletar todos os sensores para realizar testes, pois além da dificuldade de acesso às áreas em que os sensores atuam, as atividades de monitoramento teriam que sofrer uma pausa até o término do teste. A solução mais comum nestes casos é realizar os testes de *software* e *hardware* separadamente, utilizando-se simulação.

Contudo, simular separadamente as componentes não garante o funcionamento de sua integração. Em sistemas embarcados é comum que *software* e *hardware* trabalhem em conjunto para desempenhar uma tarefa. Aliás, grande parte das falhas nos testes de um *software* embarcado são oriundos da integração de componentes heterogêneos (SEO et al., 2007).

Uma das alternativas para se atenuar este efeito é utilizar emulação completa do sistema, que possibilita a visualização do estado do *software*

além de permitir o controle de sua execução (KI et al., 2008). Desta maneira, é possível analisar a intersecção entre componentes e monitorar a execução dos testes através da tabela de símbolos.

O processo de teste para sistemas heterogênicos é uma tarefa trabalhosa, principalmente quando não se sabe qual ambiente de implantação será utilizado (KARMORE; MABAJAN, 2013). Existem várias alternativas para ajudar na tarefa e aprimorar a qualidade dos testes, como utilizar oráculo como um repositório de testes (PRIYA; MANI; DIVYA, 2014), selecionar *checkpoints* baseados na interfaces dos componentes (KI et al., 2008), módulos de pré-processamento com análise de código fonte (KOONG et al., 2012), entre outros.

Sabe-se que o processo de depuração também precisa de ferramentas específicas para auxiliar no monitoramento da execução de um programa. O sucesso das técnicas que focam em automação da depuração não dependem apenas dos algoritmos ou dados coletados, mas também da estabilidade fornecida pelo ambiente para desempenhar tais atividades. Por exemplo, nas ferramentas de captura e reprodução (ORSO; KENNEDY, 2005; ORSO et al., 2006; QI et al., 2011) é essencial que a etapa de captura grave todas as operações que levam ao erro.

4.1 CONFIGURAÇÃO DO AMBIENTE REMOTO

As ferramentas utilizadas para a construção deste ambiente remoto podem ser substituídas por qualquer outra equivalente, contudo, a configuração completa será tecnicamente apresentada para que seja possível a reprodução do experimento.

O ambiente compartilhado proposto utiliza o *QEMU* para emular a máquina com a aplicação alvo a partir de outra máquina, usando tradução dinâmica. Desta forma torna-se possível utilizar-se um computador pessoal para testar aplicações compiladas para algum outro sistema embarcado. O *GNU Project Debugger* (GDB) encaixou-se no papel de depurador, pois nele é possível especificar qualquer regra que possa afetar o comportamento do SUT de maneira estática.

A integração da execução dos testes e da depuração é particular para cada máquina hospedeira e alvo, portanto, talvez alguns passos aqui apresentados devam ser adaptados para a arquitetura alvo.

A Figura 3 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:

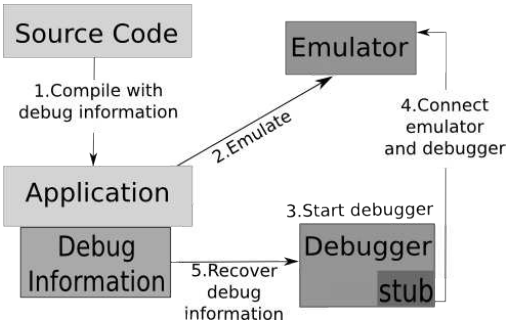

Figura 3 – Atividades de integração entre emulador e depurador

1. Compilar com informações de depuração

Este passo realiza uma compilação simbólica e sem otimizações. A otimização correta é importante para a execução do *software*, mas quando o foco é a depuração o ideal é que não haja muita diferença entre o código fonte e as instruções geradas.

Como entrada é necessário o código fonte do *software* e como saída esperada encontra-se o código compilado com informações de depuração. O *GNU Compiler Collection* (GCC) realiza esta atividade quando acrescentada a opção $-g$ para compilar.

2. Executar o sistema utilizando um emulador

É um passo necessário para executar a *software* na arquitetura alvo correta. É importante ressaltar que o *software* a ser emulado não deve iniciar a execução antes que o depurador esteja conectado. Caso isso ocorra, não será possível inspecionar todos os detalhes da execução.

Para executar este passo utilizamos o $QEMU$, que deve ser inicializado com os argumentos $-s-S$. A primeira opção ativa o *stub* para conectar um depurador, a fim de abrir a comunicação entre o emulador e o depurador. A opção $-S$ é utilizada para forçar que o $QEMU$ espere a conexão do depurador antes da inicialização do sistema. Por exemplo, se uma aplicação compilada com informações de depuração ($app.img$) imprime algo na tela ($stdio$), a chamada do $QEMU$ deve ser semelhante à: $qemu-[arch]\ -serial\ stdio\ -fda\ app.img\ -s\ -S$

3. Conectar-se com o depurador

Esta conexão é importante para fornecer ações ao usuário, tais como: executar o programa passo a passo, pausar/reiniciar a execução, entre outras. Para fornecer a estabilidade necessária para a infraestrutura

gerada, a conexão do depurador será realizada de maneira remota.

Para que a depuração seja remota a sessão do depurador deve ser inicializada em um ambiente separado. Então, para se conectar ao depurador ao *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. Utilizando-se o GDB, a tela será iniciada a partir do comando: *target remote* [*endereço_alvo*] : [*porta_alvo*]

4. Recuperar as informações de depuração

O arquivo gerado no primeiro passo 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. Então este passo é importante para ajudar os desenvolvedores a encontrarem erros.

O arquivo usado para manter as informações de depuração deverá ser informado para o GDB usando o comando: *file* [*caminho_do_arquivo*]

5. Encontrar origem dos erros

Encontrar e corrigir erros é uma atividade 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).

Após a definição da infraestrutura compartilhada para testes e depuração também é necessário dividir as funções para cada uma das atividades. A ideia é utilizar o emulador para a automação da execução do *software* e de seus casos de teste. A depuração será iniciadas somente no caso de sucesso em um teste, ou seja, quando um erro é detectado.

4.2 REFLEXÃO SOBRE O USO DO AMBIENTE COMPARTILHADO

Muitas vezes as configurações do SUT devem ser alteradas para poder executar um tipo específico de teste ou melhorar a eficácia da depuração. Estas adaptações podem influenciar no funcionamento normal do SUT e ocultar determinados tipos de erros.

A presente proposta de ambiente compartilhado não é uma exceção e, inclusive, um dos primeiros passos para utilizar o ambiente de maneira

adequada está em executar uma compilação simbólica do *software* e sem otimizações.

Para um compilador, a diferença entre utilizar ou não alguma otimização ativa é o foco da compilação do *software*. Ao utilizar uma compilação sem otimização, o objetivo do compilador será de reduzir o custo da compilação e da depuração a fim de sempre produzir os resultados esperados. Ao ativar alguma otimização sua função será compilar com o intuito de melhorar o desempenho e/ou o tamanho do código à custa do tempo de compilação e, possivelmente, da capacidade para depurar o programa (GCC. . . , 2014).

Uma compilação sem otimizações deixa o *software* mais preparado para extrair informações valiosas na depuração. Contudo, além de aumentar a quantidade de memória necessária, o fato de retirar as otimizações da compilação pode ocultar erros e distorcer as condições de corrida de um *software*.

As otimizações utilizadas por um SUT podem definir o sucesso do teste e da depuração e devem ser muito bem avaliadas para não gerarem resultados falsos. Para corroborar este fato, discutiram sobre como as otimizações podem influenciar *software* com múltiplas *threads* (JIA; CHAN, 2013). Ainda, existem vários estudos (EFFINGER-DEAN et al., 2012; FONSECA et al., 2010; KOUWE; GIUFFRIDA; TANENBAUM, 2014) que tentam fornecer alternativas para o teste e depuração de *software* com o mínimo de influência no sistema, para não ocultar erros de simultaneidade.

5 TROCA AUTOMÁTICA DOS PARÂMETROS DE *SOFTWARE*

Existe uma demanda crescente de desenvolvimento de novos sistemas ou adaptações em *softwares* já existentes. Novos requisitos de *software* e *hardware* surgem diariamente e estão cada vez mais complexos, fazendo com que os sistemas tenham que ser adaptados para que atendam a necessidades dos usuários (PRESSMAN, 2011).

Para que o esforço de desenvolver (ou manter) um *software* seja o mínimo possível, é essencial que seu código fonte seja bem projetado e estruturado, facilitando a adição/adaptação de funcionalidades (KERIEVSKY, 2008).

No desenvolvimento para sistemas embarcados é comum que uma especificação seja reutilizada, diferenciando apenas alguns requisitos não funcionais. Esta estratégia visa utilizar a propriedade intelectual já adquirida para reduzir o tempo para que um novo produto entre no mercado.

Normalmente o núcleo do sistema é formado por componentes previamente concebidos e verificados, que posteriormente são conectados à extensões (ZORIAN; MARINISSEN; DEY, 1998). Estas extensões não as adaptações/implementações de componentes de *software* ou *hardware* que atendem à um determinado requisito. Para cada nova extensão, um novo conjunto de testes é gerado para garantir o novo sistema.

Além do reuso, o desenvolvimento deste tipo de sistemas costuma ter o baixo consumo de recursos como um requisito essencial. Logo, meios que possibilitam tanto o reuso de código quanto a menor sobrecarga possível sobre o sistema, são desejáveis (IMMICH; KREUTZ; FROHLICH, 2006).

O teste de um sistema embarcado deve seguir a mesma linha de raciocínio, para tanto, o algoritmo de TAP procura atender estes requisitos ao extrapolar os conceitos da metodologia de projeto orientado aplicação e técnicas de abstração de dados para o universo dos testes.

Em um projeto orientado à aplicação o sistema é desenvolvido a partir de componentes especificamente adaptados e configurados de acordo com os requisitos da aplicação alvo. Ou seja, o próprio projeto do sistema oferece a opção de conter somente os componentes essenciais ao funcionamento da aplicação.

Os conceitos de ADESD na concepção do *software* podem reduzir o tempo gasto com o teste e a depuração do sistema, pois a partir deste tipo de projeto pode-se considerar que a entrada do sistema já está simplificada, sendo equivalente à técnicas de depuração delta. Desta forma, não é necessário utilizar técnicas como, por exemplo, partições do código para diminuir a complexidade do sistema.

Ademais, no desenvolvimento de *software* é importante que cada funcionalidade significativa seja implementada em apenas um lugar no código fonte. Caso existam funções semelhantes espalhadas pelo código, é interessante e benéfico combiná-las em uma única abstração e derivar suas peculiaridades em funções diferentes (ABELSON; SUSSMAN, 1996).

O reuso do projeto do sistema embarcado permite que todo o sistema seja reaproveitado, exigindo apenas um conjunto de testes complementar para cada extensão anexada. A abstração de casos de teste deve ser composta por testes da própria abstração e testes relacionados ao comportamento específico de cada requisito não funcional.

O algoritmo de TAP foi desenvolvido para aproveitar estas vantagens. Assim, cada SUT que possua um conjunto de requisitos que possam ser refinadas de acordo com o propósito da aplicação pode utilizar TAP para trocar automaticamente as configurações por uma abstração equivalente.

5.1 ALGORITMO DE TROCA DE PARÂMETROS

O algoritmo de troca de parâmetros foi idealizado de acordo com os conceitos de abstrações de dados e com foco no desenvolvimento e teste de sistemas embarcados.

A ideia é poder aplicar um único conjunto de testes para todas as implementações que seguem uma mesma especificação, especializando apenas a configuração desejada. Desta forma, é possível executar o mesmo conjunto de testes para atender uma maior variabilidade de configurações um componente de *software* ou *hardware*.

No Algoritmo 1 são apresentados os passos realizados a partir do momento em que se tem um SUT até o retorno do relatório para o usuário. A entrada do algoritmo é o arquivo de configuração que possui o caminho do SUT e de sua configuração - que neste caso são os *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.

Vale lembrar que o algoritmo não leva em conta a semântica da aplicação os das trocas realizadas. Se o aquivo de configuração não trouxer sugestões de propriedades a serem modificadas e dos valores semanticamente válidos, a ferramenta apenas selecionará valores estruturalmente válidos.

Algoritmo 1: Algoritmo de troca dos parâmetros de configuração

```
   Entrada: arquivo
   Saída: relatório
 1 propriedades ⇐ pegarPropriedadeDoArquivo(arquivo);
 2 se o arquivo possui valor para configuração então
 3     para cada configuração no arquivo faça
 4         linha ⇐ pegarConfiguracao(configuração, propriedades);
 5         para cada valor para configuração faça
 6             novaPropriedade⇐ modificarValorPropriedade(linha,
               propriedades) ;
 7             novaAplicacao⇐ compilar(aplicação,
               novaPropriedade) ;
 8             relatório⇐ relatório + emular(novaAplicacao) ;
 9         fim
10     fim
11 senão
12     se o arquivo possui número máximo de tentativas então
13         numMaxTentativas⇐ pegarTamanhoMaximo(arquivo);
14     senão
15         numMaxTentativas⇐ gerarValorAleatorio();
16     fim
17     enquanto tentativas < numMaxTentativas faça
18         linha⇐ gerarValorAleatorio();
19         novaPropriedade⇐ modificarValorPropriedade(linha,
           propriedades);
20         novaAplicacao⇐ compilar(aplicação, novaPropriedade);
21         relatório⇐ relatório + emular(novaAplicacao);
22     fim
23 fim
24 retorne relatório;
```

5.2 DETALHES DA IMPLEMENTAÇÃO

TAP é a ferramenta que implementa o algoritmo de troca de parâmetros de execução e utiliza o ambiente compartilhado para realizar a automação da execução dos testes e depuração de sistemas embarcados.

A implementação do algoritmo deve utiliza como base um sistema operacional com uma grande capacidade de configuração do sistema. O EPOS é 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 (FRÖHLICH, 2001). Além disso, possui vários projetos industriais e acadêmicos que o utilizam como base [1].

Este sistema operacional foi concebido com ADESD e 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 gerada possui um arquivo próprio de configuração de abstrações para definir o seu comportamento. A Figura 4 mostra um trecho desta configuração, 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 4 – Trecho do *trait* da aplicação DMEC.

Para melhorar a usabilidade da ferramenta, é possível definir um arquivo de configuração com as informações necessárias para executar os testes unitários e de tipagem. O uso de arquivos XML para definir as configurações de teste se deu pela facilidade e legibilidade para definir todas as regras necessárias para executar o algoritmo e, além disso, também é facilmente interpretado pelo computador.

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

A Figura 5 traz um exemplo do arquivo de configuração de TAP para uma aplicação exemplo.

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

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

Figura 5 – 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 "exemplo", dentro dela deseja-se especificar duas propriedades. A primeira é identificada como *ARCH* e seus valores são semanticamente válidos para a aplicação apenas se assumirem 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 CONFIGURAÇÃO DA EXECUÇÃO

Cada configuração do teste interfere diretamente no tempo e eficácia da ferramenta. 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 a TAP, do tipo de teste e das características de aplicação.

Quando se deseja testar uma especificação bem definida, é possível determinar 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 RESULTADOS EXPERIMENTAIS E ANÁLISE

Para validar o algoritmo proposto, foi implementada uma ferramenta de automação da execução da a troca de parâmetros de configuração, que utiliza o ambiente compartilhado para a execução do teste e depuração de sistemas.

Os experimentos foram elaborados para salientar os pontos positivos da proposta e também seus pontos de melhoria.

O primeiro experimento teve como objetivo averiguar se o algoritmo era robusto o suficiente para contribuir no ciclo de desenvolvimento de ferramentas complexas. Para tal, TAP realizou a troca de parâmetros de uma ferramenta utilizada na estimativa de movimento - que é um dos passos da codificação de vídeo bruto em H.264.

Já o segundo experimento foi aplicado a desenvolvedores inexperientes na implementação e manutenção de *software* embarcado, e focou na facilidade de utilização e na efetividade dos resultados. Neste caso, o experimento foi desenvolvido como base no o código produzido por alunos do curso de bacharelado em ciências da computação. Os alunos tiveram o seu primeiro contato com o EPOS durante as aulas e seus exercícios práticos consistiam em realizavam uma manutenção evolutiva do sistema.

6.1 EXPERIMENTO 1 - CICLO DE DESENVOLVIMENTO

Este experimento foi desenvolvido com o intuito de medir a robustez de algoritmo durante o ciclo de desenvolvimento e manutenção de *software*. A aplicação escolhida para este experimento foi a DMEC, que simula um componente de estimativa de movimento distribuída.

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).

A Figura 6 apresenta a interação que ocorre na aplicação: 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.

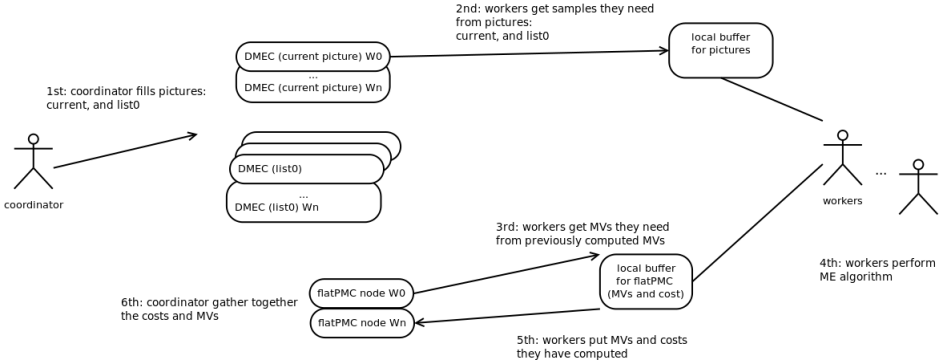

Figura 6 – Interação entre o coordenador e os trabalhadores na aplicação teste do DMEC (LUDWICH; FROHLICH, 2011).

6.1.1 Execução dos testes e depuração

A aplicação de 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.

Um dos requisitos desta aplicação é a produção de estimativas consumindo o menor tempo possível. Desta forma, a TAP foi configurada para trocar a configuração do número de trabalhadores(`NUM_WORKERS`), a fim de paralelizar o trabalho da estimativa. A configuração inicial de número de trabalhadores da aplicação era 6 e decidiu-se aumentar em dez vezes este número.

A troca de parâmetros gerou todas as configurações de `NUM_WORKERS` de 1 até 60 em nenhuma delas houve erro de compilação. O teste do limite inferior e superior são demonstrados, respectivamente, na Figura 7 e na Figura 8.

As Figuras 7 e 8 conseguiram produzir uma imagem executável após a compilação, entretanto, mesmo sem haver erro na compilação a execução com `NUM_WORKERS` igual a 60 retornou um resultado inválido. Desta forma, torna-se visível que apenas compilar o código não garante que a aplicação é livre de *bugs*.

Para os casos nos quais a execução da aplicação não foi bem sucedida TAP automaticamente aciona o depurador para que se possa desco-

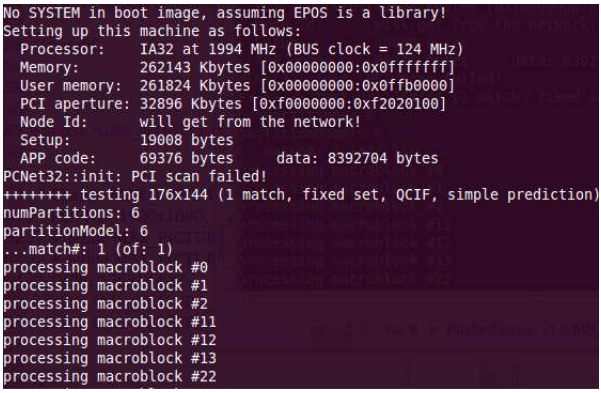

Figura 7 – Teste do DMEC com configuração NUM_WORKERS = 6

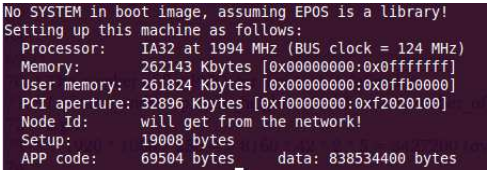

Figura 8 – Teste do DMEC com configuração NUM_WORKERS = 60

brir o porquê deste comportamento. Sendo assim, foi necessário incluir na configuração da ferramenta um arquivo com *breakpoints* para poder verificar-se o problema. Foram adicionados pontos de interrupção em cada uma das funções da aplicação, inclusive na função principal. A execução do GDB com os *breakpoints* é demonstrada na Figura 9.

Com esta configuração apenas execuções que chamaram ao menos um vez cada uma das funções foram consideradas execuções corretas da aplicação. Note-se que após o comando `continue` não houve mais nenhuma parada em qualquer uma das funções. Insto significa que para a configuração NUM_WORKERS com o valor 60 não houve nenhuma resposta da aplicação e que, inclusive, não conseguiu nem atingir a função principal.

Ainda, observando-se o relatório final emitido pela ferramenta foi possível descobrir que sempre que a configuração NUM_WORKERS apresentava um número maior que 10 a aplicação se comportava de maneira anômala.

Após a execução da ferramenta e análise do relatório foi possível determinar que existia um limite máximo de trabalhadores, informação esta que ajudou no desenvolvimento do componente. A partir desta descoberta foi possível maximizar o paralelismo das estimativas sem que houvesse efeitos

```
(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 9 – Depuração do DMEC com a configuração NUM_WORKERS = 60

colaterais no funcionamento da aplicação.

Entretanto, a qualidade da informação de retorno é inerente à qualidade de informação à priori que é repassada na configuração de TAP. A Figura 10 apresenta um trecho de relatório com algumas configurações geradas.

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

Em casos como o do teste 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. Relatórios gerados com mais dados tendem a ser mais concisos e menos repetitivos, já os oriundos de testes aleatórios tendem a uma menor organização e maior redundância nas informações.

Outro ponto de interesse científico encontra-se na razão entre o consumo de tempo para executar os testes versus a qualidade da informação que

pode ser extraída. As Figuras 11 e 12 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.

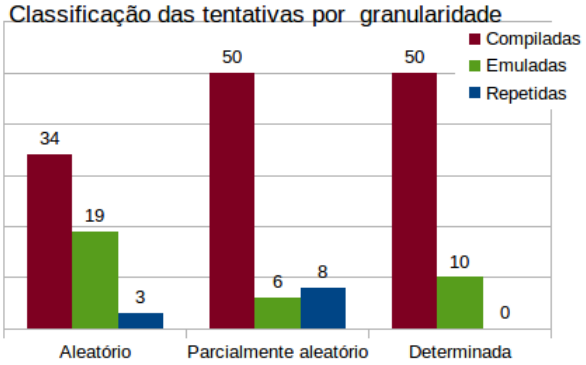

Figura 11 – Classificação das tentativas realizadas versus a configuração da granularidade.

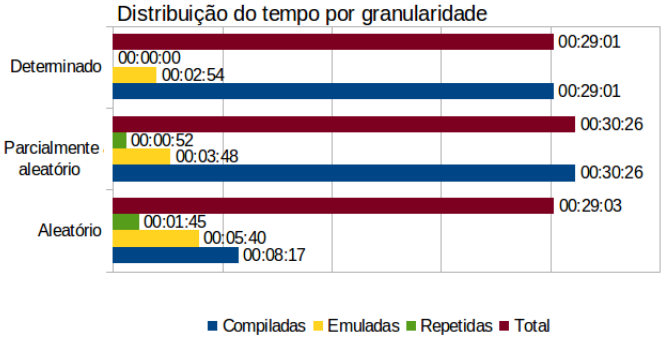

Figura 12 – Classificação das tentativas realizadas versus o consumo de tempo.

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 li-

gada 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 13, 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.

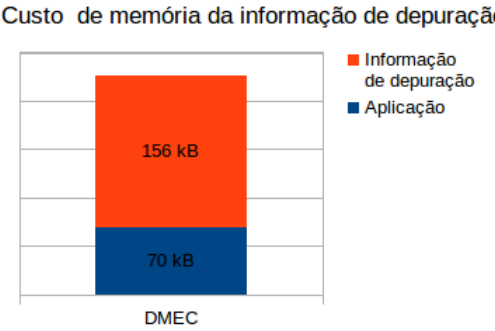

Figura 13 – Consumo de memória extra para armazenar as informações de depuração.

6.2 EXPERIMENTO 2 - FACILIDADE DE USO E EFETIVIDADE DOS RESULTADOS

Este experimento foi elaborado para verificar a facilidade de utilização do algoritmo, do ambiente compartilhado de teste e depuração de *software* e dos resultados apresentados pela ferramenta.

Para verificar a facilidade do uso o experimento deveria ser aplicado à desenvolvedores inexperientes na produção de *software* embarcado. A ideia é apresentar a ferramenta para desenvolvedores inexperientes e configurá-la para que o algoritmo de troca de parâmetros seja capaz de apontar que um determinado teste apresentou um comportamento errado por causa de determinada configuração.

TAP foi avaliada através do código fonte produzido por estudantes do curso de bacharelado em ciências da computação e que estavam cursando a disciplina de sistemas operacionais II. A escolha da turma se deu devido

ao fato de ser uma disciplina com muitos exercícios práticos, os quais são baseados em um sistema operacional (EPOS) cuja disponibilidade é controlada por senha, ou seja, sabe-se de antemão quais estudantes já tiveram contato com o sistema.

No início desta disciplina os alunos tem acesso a um EPOS incompleto e cada exercício realizado tem o intuito de implementar uma parte deste *software*. Todos os exercícios tem um escopo fechado e sabe-se quais testes devem ser executados corretamente para cada uma das fases.

O EPOS permite que cada aplicação gerada possua um arquivo próprio de configuração de abstrações para definir o seu comportamento. Dentre estas configurações encontram-se definições como, por exemplo, o tipo de escalonamento, que podem afetar um determinado comportamento da aplicação sem modificar seu propósito.

Se existem várias configurações válidas de uma mesma aplicação, todas elas devem ser atendidas pelas soluções dos exercícios. Entretanto, devido à inexperiência, alguns alunos não utilizam esta troca de parâmetros como ferramenta de apoio.

Para fornecer dicas sobre as configurações mais importantes, TAP foi configurada para realizar a troca das seguintes configurações: tipo de escalonamento de processos, o quantum de tempo e tamanhos da *stack* e *heap*. O arquivo de configuração desenvolvido para a disciplina pode ser observado na Figura 14.

6.2.1 Os exercícios e a execução dos testes

Durante a disciplina os alunos precisam realizar vários exercícios práticos, para que aos poucos atingir a maturidade necessária para realizar um projeto final.

6.2.1.1 O projeto final

O projeto final consistem em transformar o EPOS em um *microkernel*. Deve-se então realizar uma evolução no sistema, mas agora o escopo aumenta e todos os testes devem ser executados com sucesso. O sistema final deverá suportar a criação de multiplos processos e garantir o princípio de não interferência entre as *threads* e delas para com o *kernel*.

Como este projeto é extenso, ele foi separado em várias entregas parciais, a fim de acompanhar o desempenho dos alunos. Os dados retirados da execução dos testes do projeto final podem ser observados na Figura X.

```
<test>
   <configuration>
     <trait scope="application" id="STACK_SIZE">
         <value>512</value>
         <value>1024</value>
         <value>2048</value>
         <value>3072</value>
         <value>4096</value>
      <trait>

      <trait scope="application" id="HEAP_SIZE">
         <value>512</value>
         <value>1024</value>
         <value>2048</value>
         <value>3072</value>
         <value>4096</value>
      <trait>

      <trait scope="System" id="STACK_SIZE">
         <value>512</value>
         <value>1024</value>
         <value>2048</value>
         <value>3072</value>
         <value>4096</value>
      <trait>

      <trait scope="System" id="HEAP_SIZE">
         <value>512</value>
         <value>1024</value>
         <value>2048</value>
         <value>3072</value>
         <value>4096</value>
      <trait>

      <trait scope="System" id="QUANTUM">
         <value>10000</value>
         <value>20000</value>
         <value>50000</value>
         <value>100000</value>
         <value>200000</value>
         <value>500000</value>
      <trait>

     <trait scope="Thread" id="QUANTUM">
         <value>10000</value>
         <value>20000</value>
         <value>50000</value>
         <value>100000</value>
         <value>200000</value>
         <value>500000</value>
      <trait>

     <trait id="Scheduling_Criteria">
         <value>"RR"</value>
         <value>"FCFS"</value>
      <trait>
   </configuration>
<test>
```

Figura 14 – Configuração de TAP para as aplicações do EPOS da disciplina.

7 ANÁLISE QUALITATIVA DAS 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 inspiradas na pesquisa de Antonia Bertolino (BERTOLINO, 2007), no qual são explicitados os conceitos mais relevantes para a área de testes de *software*, separados em realizações passadas e em metas ainda não atingidas.

7.1 METAS

Dentre as metas para a pesquisa em teste de *software*, as relacionadas diretamente com o escopo deste trabalho são: teoria de testes universal, modelagem baseada em teste e testes 100% automatizados. Cada meta é composta por um conjunto de desafios essenciais para o avanço do estado da arte.

7.1.1 Teoria de teste universal

Esta teoria propõe a adoção de um padrão coerente para a criação de modelos ou técnicas de teste. A partir desta padronização será possível averiguar os pontos fortes e as limitações de cada opção e, consequentemente, conseguir optar racionalmente a melhor opção para cada caso.

7.1.1.1 Desafio das hipóteses explícitas

Com a exceção de algumas abordagens formais, normalmente os testes são baseados 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 elucidar o porquê de observamos algumas execuções.

7.1.1.2 Desafio da Eficácia do teste

Este desafio aborda o porquê, como e quantos testes são necessários para descobrir um determinado tipo de erro. Pois para estabelecer uma teoria útil para a elaboração e execução dos testes é preciso avaliar a eficácia dos critérios para elaboração teste e sua execução.

É imprescindível conseguir analisar as teorias existentes e das novas técnicas que estão surgindo. Apesar de não haver um padrão para aferir a eficácia, a controvérsia convencional entre a técnica proposta versus técnicas aleatórias é amplamente utilizada até pelos métodos mais sofisticados.

7.1.1.3 Desafio dos testes de composição

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.

Este desafio está relacionado ao teste de sistemas complexos, por focar em entender como podemos reutilizar os resultados da decomposição e quais conclusões podem ser inferidas para o sistema a partir destes resultados.

7.1.1.4 Desafio das evidências empíricas

Na pesquisa de teste de *software*, estudos empíricos são essenciais para avaliar as técnicas e práticas propostas, entender como elas funcionam e aperfeiçoá-las. Infelizmente, grande parte do conhecimento de técnicas de teste existentes são 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, contexto e tema abordado.

O desafio das evidências empíricas procura fomentar a consciência de unir forças para aprimorar os experimentos está se espalhando e já conta com iniciativas como a construção de repositórios de dados compartilhados e de bancos de dados de ensaios experimentais.

7.1.2 Modelagem baseada em teste

A concepção tradicional de testes de *software* é desenvolver os casos de teste à partir do estudo do *software* e procurar alternativas para melhor explorá-lo.

Esta meta defende que um *software* só pode ser efetivamente testado se ele for desenvolvido à partir de um modelo. A ideia é realizar uma inversão de valores, ou seja, migrar de uma geração de casos de testes baseado em modelos para uma modelagem do sistema baseada em teste.

7.1.2.1 Desafio dos testes baseados em modelos

Em sistemas complexos e heterogêneos é comum que se utilize mais de um tipo de modelagem de *software* para definir suas funcionalidades. A meta desse desafio é garantir que os modelos resultantes de diferentes paradigmas possam ser expressos em qualquer notação e serem perfeitamente integrados dentro de um único ambiente.

Um dos casos promissores de teste baseados em modelos é o ensaio de conformidade, cujo objetivo é verificar se o SUT está em conformidade com a sua especificação, considerando alguma relação definida no modelo.

7.1.2.2 Desafio dos testes baseados em anti-modelo

Este desafio surgiu da hipótese de que pode não existir um modelos do *software*. Isto pode acontecer, por exemplo, em casos em que um modelo é originalmente criados, mas não há manutenção da correspondência entre o modelo e a implementação, tornando-se obsoleto.

Este desafio foca em abordagens de teste anti-modelo. Ou seja, em coletar informações da execução do programa e tentar sintetizar as propriedades do sistema em um novo modelo ao invés de elaborar um plano de teste e comparar os resultados do teste com o modelo original.

7.1.2.3 Desafio dos oráculos de teste

Um oráculo é uma heurística que pode emitir um veredicto de aprovação ou reprovação das saídas de um determinado teste. Este desafio pretende solucionar o problema de derivar os casos de teste e de como decidir se um resultado do teste é aceitável ou não.

A precisão e a eficiência dos oráculos afeta muito custo do teste, pois não é aceitável que falhas nos testes passem despercebidos, mas por outro lado não é desejável que hajam muitos falsos positivos, que desperdiçam recursos importantes.

7.1.3 Testes 100% automáticos

A automação total dos testes depende de um ambiente poderoso. Ele deve automaticamente providenciar a instrumentação do *software*, a geração e recuperação do suporte necessário (ex. *drivers*, *stubs*, simuladores, emuladores), ser responsável pela geração automática dos casos de teste mais adequados para o modelo, executá-los e, finalmente, emitir um relatório sobre o ensaio executado.

Ainda existem muitos desafios a serem solucionados para atingir este nível de automação, contudo, os que mais se relacionam com o presente trabalho são a geração de dados de entrada para o teste, as abordagens específicas de domínio e a execução de testes *online*.

7.1.3.1 Desafio da geração de dados de entrada

A geração de dados de entrada para os testes sempre foi um tópico de pesquisa muito ativo e utilizados academicamente. Entretanto, este esforço produz um impacto limitado na indústria, onde a atividade de geração de teste permanece em grande parte manual.

Os resultados mais promissores são as abordagem baseada em modelo e a geração aleatória acrescida de alguma técnica com inteligência. Desta forma, ainda se faz necessária uma técnica que possa ser utilizada de maneira mais abrangente.

7.1.3.2 Desafio das abordagens de teste específicas para o domínio

O atual estado da arte aponta para a necessidade executar a fase de testes com abordagens específicas de domínio.

O intuito deste desafio é encontrar métodos e ferramentas específicas de domínio e poder aprimorar a automação de teste. Neste sentido, alguns trabalhos já conseguiram demonstrar a extração automática de requisitos para o teste a partir de um modelo escrito em uma linguagem fortemente tipada e específica de um domínio.

7.1.3.3 Desafio dos testes *online*

O desafio de testes *online* focam na ideia de monitorar o comportamento de um sistema em pleno funcionamento, com o *software* executando e recebendo todas as interferências reais.

É um desafio importante e complexo, pois nem sempre é possível realizar este tipo de teste, especialmente para aplicações embarcadas implantados em um ambiente de recursos limitados, onde a sobrecarga exigida pela instrumentação de teste não poderia ser viável.

7.2 ANÁLISE DA FERRAMENTA PROPOSTA

A partir da contextualização das características importantes para a evolução do estado da arte, agora já é possível realizar uma análise qualitativa de TAP, ressaltando as contribuições e melhorias da ferramenta. As ferramentas e técnicas propostas nos trabalhos relacionados também serão analisadas sob o mesmo ponto de vista.

7.2.1 Meta da teoria de testes universal

A primeira meta abordada, a teoria de testes universal, possui desafios para impulsionar as pesquisas em projeto de testes de *software*.

Apesar do foco de TAP permanecer na execução dos testes, sua concepção e a automação da execução possuem pontos em comum. O desafio de hipóteses explícitas, por exemplo, é um critério originalmente abordado no projeto de testes, mas que possui grande relevância para a execução, pois fazem parte dos dados de entrada.

TAP foi analisada sob a meta de teoria de teste universal e seus resultados encontram-se na Tabela 1.

Nesta primeira análise observa-se que a ferramenta atende a todos os desafios proposto, exceto o desafio de evidências empíricas. Não foi possível realizar a execução de testes de um repositório devido à utilização da modelagem ADESD, que possui especificidades que ainda não estão contempladas nos *testbeds* disponíveis. Um ponto de melhoria seria compartilhar o modelo de testes aplicado no Capítulo 6 em outros repositórios, aumentando o volume de dados para a avaliação de outras propostas.

Considerando as mesmas metas e objetivos, vários dos trabalhos relacionados atendem os mesmos quesitos que TAP e com a mesma intensidade. Dentre a técnicas que mais se assemelha à TAP estão as ferramentas

Tabela 1 – Análise de TAP para a meta de teoria de teste universal

Hipóteses explícitas	As hipóteses podem ser **parcialmente** explicitadas através do arquivo de configuração, onde é possível importar definições semi-formais.
Eficácia do teste	Cobertura **total**, onde a eficácia é testada de forma quantitativa pela controvérsia convencional entre a técnica proposta versus a técnica aleatória.
Testes de composição	Testes **parcialmente** atendidos, pois cada componente pode ser testado como uma aplicação menor e a integração destes componentes forma a aplicação real. Entretanto, a composição dos componentes é apenas simulada.
Evidências empíricas	O algoritmo, sua implementação e o conjunto de testes utilizados estão liberadas para o uso de terceiros, mas devido à especificidades na modelagem do sistema **não foi possível** utilizar testes de repositórios já existentes.

de depuração como as de depuração estatística e depuração delta. Isto ocorre porquê ferramentas que implementam estas técnicas não são voltadas para a geração de casos de teste e utilizam-se de hipóteses explícitas como dados de entrada para posteriormente realizar a depuração. A eficácia da execução dos testes/depuração são normalmente aferidas levando-se em conta a execução das ferramentas e confrontando-as com modelos aleatórios.

A análise de *Justitia* também aponta semelhanças na declaração parcial de hipóteses explícitas para os dados de entrada de teste - que são representadas pelas interface do sistema - e na eficácia dos testes. A vantagem de *Justitia* está nos testes de composição, que não são amplamente contemplados em TAP por ser uma características voltada para a geração de casos de testes.

Já a comparação de TAP com *ATEMES* as características só se distanciam nos desafios de eficácia do teste e testes de composição. Ambas ferramentas possuem estratégias diferentes, visto que *ATEMES* foca em explorar sistematicamente a componentização da própria ferramenta e a robustez no teste do *software* em detrimento da avaliação da eficácia dos testes.

A única técnica avaliada que não apresenta hipóteses explícitas é a captura e reprodução. Isto ocorre porque a entrada das ferramentas não são as hipóteses, e sim o próprio executável, ou seja, observa-se o sistema através

da captura de toda a sua execução, não sendo necessário observar execuções relacionadas à uma determinada hipótese.

A Tabela 2 apresenta um resumo da análise comparativa entre TAP e as ferramentas correlatas no quesito teoria de teste universal.

Tabela 2 – Resumo comparativo do suporte para a meta de teoria de teste universal

Técnicas	**Suporte para o desafio**			
	Hipóteses explícitas	**Eficácia do teste**	**Testes de composição**	**Evidências empíricas**
TAP	Parcial	Sim	Parcial	Não
Justitia	Parcial	Sim	Sim	Não
ATEMES	Parcial	Parcial	Sim	Não
Execução de testes por sistema alvo	Não	Parcial	Parcial	Não
Partição do *software*	Parcial	Sim	Sim	Não
Depuração estatística	Parcial	Sim	Parcial	Não
Depuração por delta	Parcial	Sim	Parcial	Não
Captura e reprodução	Não	Parcial	Parcial	Não

Um aspecto que ficou bastante realçado que nenhuma das técnica contemplou o desafio das evidências empíricas. Apesar de grande parte dos autores disponibilizam seus trabalhos para o púbico em geral, os dados derivados das pesquisas ainda não são maduros o suficiente parar desenvolver o estado da arte. Ou seja, grande parte dos dados existentes são originados de impressões e deduções dos próprios autores e, muitas vezes, desprovidos de fundamento formal.

7.2.2 Meta da modelagem baseada em testes

Devido à falta de maturidade na produção de sistemas, hoje há um foco muito grande no desenvolvimento do *software*, e as atividades de teste e depuração são apenas subprodutos que ajudam na confiabilidade do sistema.

Neste contexto, a meta de modelagem baseada em testes se torna muito importante, pois defende um *software* só pode ser completamente testado caso haja um planejamento para a geração dos testes e que sejam originados à partir de um modelo definido.

Uma análise das contribuições de TAP na meta de modelagem baseada

em testes será apresentada na Tabela 3.

Tabela 3 – Análise de TAP para a meta de modelagem baseada em testes

Teste baseado em modelos	Atualmente a ferramenta possui uma contribuição **limitada** à agregação de outros modelos e prevê apenas o ensaio de conformidade a partir de uma configuração de teste.
Teste baseado em anti-modelo	Apesar de não ser o foco deste trabalho, o relatório fornecido por TAP apresenta todas as características e os valores trocadas em tempo de execução. Estes dados podem ser utilizados para realizar uma modelagem do sistema, portanto a ferramenta **possui** suporte para o anti-modelo.
Oráculos de teste	TAP **não** contém um oráculo de teste, uma vez que não faz parte deste trabalho a geração de casos de teste.

Tabela 4 – Análise de TAP para a meta de modelagem baseada em testes

Todos os desafios da meta de modelagem baseada em testes são relevantes para a área de testes de *software*, contudo, grande parte dos desafios são contemplados de maneira parcial em TAP por distanciar-se muito dos objetivos específicos deste trabalho. Satisfazer esta meta é um ponto de melhoria que deve ser analisado para implementação futura.

Durante o comparativo entre as ferramentas foi possível identificar que todos os trabalhos abordam de alguma forma o desafio de testes baseados em modelos e anti-modelos, exceto *Justitia*, que só suporta os testes baseados em modelos. *Justitia* utiliza as interfaces como modelos para a construção dos casos de teste e também como *checkpoint* das atividades do *software*, não sendo possível utilizar outra abordagem de verificação do sistema.

Dentre todas as técnicas analisadas, apenas duas contemplaram a meta por completo, apresentando soluções para todos os desafios propostos: Captura e reprodução e *ATEMES*.

Na Captura e reprodução, o padrão é o teste baseado em anti-modelos, pois toda a execução do sistema é gravada e utilizada para o teste e depuração do sistema. Embora seja menos comum, algumas ferramentas da técnica ainda oferecem o teste baseado em modelo, que são mesclados os dados do anti-modelo para fornecer um resultado melhor na identificação dos *bugs*.

Já *ATEMES* se destaca pela quantidade de tipos de teste que podem ser gerados pelo módulo `PRPM` da ferramenta, sendo que a base destes casos de teste pode ser vir de modelos bem definidos, anti-modelos, agregação de modelos ou até de maneira aleatória. O oráculo de *ATEMES* processa as informações de execução do sistema e repassa informações para vários

módulos, por exemplo, o módulo POPM recebe um conjunto de linhas do código fonte apontadas como prováveis geradoras de erros.

Outra técnica que se destaca pelo oráculo de testes é a Execução de testes por sistema alvo. Além de permitir uma alimentação à partir de execuções consideradas corretas do *software*, recentemente uma das implementações da técnica adaptou-a com um oráculo composto por uma rede neural com *back-propagation*.

Depuração por delta e Depuração estatística são técnicas de depuração e por isso não apresentam um oráculo de testes. Algo semelhante ocorre com a partição de código, pois sua principal característica é particionar o código, respeitando o modelo (ou anti-modelo) do *software*. Este particionamento pode ocorrer de várias formas, mas não é um oráculo quem define qual algoritmo utilizar.

A Tabela 5 apresenta um resumo da análise comparativa entre as ferramentas e técnicas correlatas para os desafios propostos pela meta de modelagem baseada em teste.

Tabela 5 – Comparativo qualitativo do suporte para a meta de modelagem baseada em teste

Técnicas	**Suporte para o desafio**		
	Teste baseado em modelos	**Teste baseado em anti-modelos**	**Oráculos de teste**
TAP	Parcial	Sim	Não
Justitia	Parcial	Não	Parcial
ATEMES	Sim	Sim	Sim
Execução de testes por sistema alvo	Sim	Não	Sim
Partição do *software*	Sim	Sim	Não
Depuração estatística	Sim	Sim	Não
Depuração por delta	Sim	Sim	Não
Captura e reprodução	Parcial	Sim	Parcial

Analisando o comparativo é possível identificar que as técnicas e ferramentas que possuem a geração de testes dispõem de um oráculo. Esta preocupação é apropriada para que a geração dos testes seja efetiva para encontrar os erros e melhorar cada vez mais a qualidade do código.

Outra perspectiva relevante é a utilizada pelas ferramentas de depuração, que responsabilizam-se por depurar um *software* e descobrir se ele é correspondente a um determinado modelo. Quando não há um modelo disponível,

estas ferramentas estão preparadas para comparar as diversas execuções do próprio *software* e efetuar a análise sob os dados obtidos do próprio SUT.

7.2.3 Meta dos testes 100% automáticos

A meta dos testes completamente automatizados permite um controle de qualidade no desenvolvimento e manutenção de um *software*. É de difícil implantação, mas uma vez que a automação esteja em pleno funcionamento é possível manter a confiabilidade do sistema de maneira rápida e eficiente.

A Tabela 6 mostra a relação de TAP com a meta de automação completa da execução de testes de *software*, em que nota-se níveis de automação em todos o desafios propostos.

Tabela 6 – Análise de TAP para a meta de automação dos testes

Geração de dados de entrada	O suporte à geração de dados é **parcial**, pois a informação inicial dos dados de entrada são fornecidos pelo usuário da ferramenta. No caso do teste não ser determinado pelo usuário, a ferramenta inclui dados iniciais aleatórios, sem adição de inteligência. Também existe a opção de retroalimentação.
Abordagens específicas para o domínio	TAP consegue importar configurações específicas de domino para realizar a execução de testes, contemplando de forma **parcial** a extração de dados relacionados ao domínio da aplicação.
Testes *online*	Este diferencial é apresentado pela ferramenta através do ambiente integrado de teste e depuração, com suporte **total** aos testes durante à execução da aplicação. Todavia, quando a execução da aplicação ocorre em ambiente emulado o suporte à testes *online* é considerado parcial.

Para ser considerada uma ferramenta 100% automática TAP, ainda deve-se investir em técnicas para receber a entrada dos dados, seja a partir da extração de valores do modelo ou através de algum tipo de inteligência artificial. Ao integrar TAP ferramentas como, por exemplo, a *ADESDTool* (CANCIAN, 2011) pode-se fornecer melhores configurações para o próprio sistema embarcado.

A geração de dados de entrada é uma característica que muitas vezes é preterida por ferramentas que não realizam a geração dos casos de teste. Técnicas como a Partição de *software* e Captura e reprodução empregam o modelo do *software* como dado de entrada para o teste/depuração e não são

capazes incluir mutabilidade coerente ao modelo.

Conjuntamente, a abordagem específica de domínio também exerce bastante influência para uma automação total dos testes. Existem trabalhos totalmente focados para uma abordagem específica de domínio, como é o caso da Execução de testes por sistema alvo. Nesta técnica considera que cada sistema embarcado possui propriedades que o definem e, portanto, devem ser levadas em conta na escolha do modelo de testes a ser utilizado.

A Tabela 7 apresenta uma análise comparativa entre as ferramentas e técnicas correlatas para os desafios propostos através de testes 100% automatizados.

Tabela 7 – Comparativo qualitativo do suporte para a meta de testes 100% automáticos

Ferramentas	**Suporte para o desafio**		
	Geração de entradas	**Abordagem específica para domínio**	**teste *online***
TAP	Parcial	Parcial	Sim
Justitia	Sim	Não	Sim
ATEMES	Sim	Não	Sim
Execução de testes por sistema alvo	Parcial	Sim	Sim
Partição do *software*	Não	Parcial	Sim
Depuração estatística	Não	Parcial	Não
Depuração por delta	Parcial	Parcial	Não
Captura e reprodução	Não	Não	Sim

Esta meta deixa em evidência que o ambiente integrado de TAP fornece uma grande utilidade para a conseguir realizar tanto o teste quando a depuração *online*. Isto não ocorre com as outras ferramentas de depuração, que deixam a desejar no desafio de execução *online*.

8 CONCLUSÕES

Neste trabalho foi introduzida TAP, uma ferramenta para a troca automática de parâmetros de configuração e apresentado um roteiro para a geração de um ambiente de desenvolvimento de aplicações embarcadas baseadas em requisitos de *hardware* e *software* específicos.

O ambiente de desenvolvimento integrado fornece independência em relação à plataforma física de destino. Desta forma, os desenvolvedores não precisam gastar tempo compreendendo uma nova plataforma de desenvolvimento sempre que alguma característica do sistema embarcado for atualizada. Este é um passo importante, pois alguns sistemas embarcados podem não ser capazes de armazenar os dados adicionais necessários para apoiar a depuração.

Os experimentos realizados apontaram valores quantitativos do tempo consumido para realizar os teses e da efetividade dos ensaios. Foi confirmada que a eficácia do algoritmo está intimamente ligada à efetividade da configuração dos valores e da granularidade apresentadas à ferramenta.

Foram avaliados os impactos inerentes ao uso da ferramenta para verificar se existiam pontos de melhoria, que devem ser tratados em trabalhos futuros. Dentre eles estão a redução do uso de memória e do tempo utilizados para recuperar as informações de depuração, que atualmente apresenta um aumento de mais de 500% no tamanho do código da aplicação e a uma sobrecarga de 60% para o tempo de execução do teste.

Também foi realizada a análise qualitativa da ferramenta, levando-se em conta os desafios ainda presentes no teste de *software* e metas que necessitam ser atingidas. Os resultados demonstram o comprometimento da ferramenta com o avanço do estado da arte, pois procura oferecer uma solução viável a pelo menos 80% dos desafios propostos.

A ferramenta possui resultados promissores se comparada com outras ferramentas correlatas, frequentemente apresentando soluções equivalentes às de outras ferramentas e, eventualmente, cobrindo características que não fazem parte do escopo inicial deste trabalho.

8.1 PERSPECTIVAS FUTURAS

Durante o desenvolvimento deste trabalho foram identificados pontos que podem ser otimizados em trabalhos futuros:

- Analisar a possibilidade de reduzir o custo de memória e de tempo utilizado para a execução de testes.

- Verificar formas de melhorar a configuração de TAP e dos dados de entrada do algoritmo.
- Realizar um maior número de experimentos para verificar o desempenho da ferramenta em ambientes com restrições e que apresentem testes mais complexos.
- Aprimorar a ferramenta para conseguir atingir os desafios identificados na área de teste e depuração de *software*.

O desenvolvimento deste trabalho resultou em artigos publicados em eventos, e que contribuirão para o estado da arte nas áreas de verificação de *software* e de construção de sistemas embarcados. Como perspectiva futura também encontra-se a produção de mais artigos.

REFERÊNCIAS BIBLIOGRÁFICAS

ABELSON, H.; SUSSMAN, G. J. *Structure and Interpretation of Computer Programs*. 2nd. ed. Cambridge, MA, USA: MIT Press, 1996. ISBN 0262011530.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381.

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.

BEIZER, B. *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN 0-442-20672-0.

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.

BINKLEY, D. et al. Minimal slicing and the relationships between forms of slicing. In: IEEE. *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*. [S.l.], 2005. p. 45–54.

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.

CANCIAN, R. L. *Um Modelo EvolucionÃ¡rio Multiobjetivo para ExploraÃ§Ã£o do EspaÃ§o de Projeto em Sistemas Embarcados*. 258 p. Tese (Doutorado) — Federal University of Santa Catarina, Florianópolis, 2011. Ph.D Thesis.

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

CHEBARO, O. et al. Program slicing enhances a verification technique combining static and dynamic analysis. In: *Proceedings of the 27th*

Annual ACM Symposium on Applied Computing. New York, NY, USA: ACM, 2012. (SAC '12), p. 1284–1291. ISBN 978-1-4503-0857-1. <http://doi.acm.org/10.1145/2245276.2231980>.

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>.

EBERT, C.; JONES, C. Embedded software: Facts, figures, and future. *IEEE Computer*, v. 42, n. 4, p. 42–52, 2009.

EFFINGER-DEAN, L. et al. Ifrit: Interference-free regions for dynamic data-race detection. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2012. v. 47, n. 10, p. 467–484.

FONSECA, P. et al. A study of the internal and external effects of concurrency bugs. In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. [S.l.: s.n.], 2010. p. 221–230.

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

FROLA, F. R.; MILLER, C. *System safety in aircraft acquisition*. [S.l.], 1984.

GCC : Options That Control Optimization. nov. 2014. <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.

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>.

HOPKINS, A. B. T.; MCDONALD-MAIER, K. D. Debug support for complex systems on-chip: a review. *Computers and Digital Techniques, IEE Proceedings* -, v. 153, n. 4, p. 197–207, July 2006. ISSN 1350-2387.

IMMICH, R.; KREUTZ, D.; FROHLICH, A. Resource management for embedded systems. In: *Factory Communication Systems, 2006 IEEE International Workshop on*. [S.l.: s.n.], 2006. p. 91–94.

JGRENNING, J. Progress before hardware. In: . [S.l.: s.n.], 2004.

JIA, C.; CHAN, W. Which compiler optimization options should i use for detecting data races in multithreaded programs? In: IEEE. *Automation of Software Test (AST), 2013 8th International Workshop on*. [S.l.], 2013. p. 53–56.

KARMORE, S.; MABAJAN, A. Universal methodology for embedded system testing. In: *Computer Science Education (ICCSE), 2013 8th International Conference on*. [S.l.: s.n.], 2013. p. 567–572.

KERIEVSKY, J. *Refatoração para padrões*. [S.l.]: Bookman, 2008.

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>.

KOUWE, E. V. D.; GIUFFRIDA, C.; TANENBAUM, A. On the soundness of silence: Investigating silent failures using fault injection experiments. In: *Dependable Computing Conference (EDCC), 2014 Tenth European*. [S.l.: s.n.], 2014. p. 118–129.

LEVESON, N. G. Role of software in spacecraft accidents. *Journal of spacecraft and Rockets*, v. 41, n. 4, p. 564–575, 2004.

LEVESON, N. G. Software challenges in achieving space safety. British Interplanetary Society, 2009.

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.

MISHERGHI, G.; SU, Z. Hdd: Hierarchical delta debugging. In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: ACM, 2006. (ICSE '06), p. 142–151. ISBN 1-59593-375-1. <http://doi.acm.org/10.1145/1134285.1134307>.

MYERS, G. J.; SANDLER, C. *The Art of Software Testing*. [S.l.]: John Wiley & Sons, 2004. ISBN 0471469122.

ORSO, A. et al. Isolating relevant component interactions with jinsi. In: ACM. *Proceedings of the 2006 international workshop on Dynamic systems analysis*. [S.l.], 2006. p. 3–10.

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.

POTTIE, G. J.; KAISER, W. J. Wireless integrated network sensors. *Communications of the ACM*, ACM, v. 43, n. 5, p. 51–58, 2000.

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

PRIYA, P. B.; MANI, M. V.; DIVYA, D. Neural network methodology for embedded system testing. *International Journal of Research in Science & Technology*, v. 1, n. 1, p. 1–8, 2014.

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.

ROOK, P. Controlling software projects. *Software Engineering Journal*, v. 1, n. 1, p. 7–, January 1986. ISSN 0268-6961.

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.

SRIDHARAN, M.; FINK, S. J.; BODIK, R. Thin slicing. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2007. (PLDI '07), p. 112–122. ISBN 978-1-59593-633-2. <http://doi.acm.org/10.1145/1250734.1250748>.

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.

TORRI, L. et al. An evaluation of free/open source static analysis tools applied to embedded software. In: IEEE. *Test Workshop (LATW), 2010 11th Latin American*. [S.l.], 2010. p. 1–6.

VENKATESH, G. A. The semantic approach to program slicing. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 26, n. 6, p. 107–119, maio 1991. ISSN 0362-1340. <http://doi.acm.org/10.1145/113446.113455>.

WEISER, M. Program slicing. In: *Proceedings of the 5th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 1981. (ICSE '81), p. 439–449. ISBN 0-89791-146-6. <http://dl.acm.org/citation.cfm?id=800078.802557>.

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>.

ZELLER, A. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 24, n. 6, p. 253–267, out. 1999. ISSN 0163-5948. <http://doi.acm.org/10.1145/318774.318946>.

ZELLER, A.; HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, IEEE, v. 28, n. 2, p. 183–200, 2002.

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.

ZHIVICH, M.; CUNNINGHAM, R. K. The real cost of software errors. Institute of Electrical and Electronics Engineers (IEEE), 2009.

ZORIAN, Y.; MARINISSEN, E.; DEY, S. Testing embedded-core based system chips. In: *Test Conference, 1998. Proceedings., International*. [S.l.: s.n.], 1998. p. 130–143. ISSN 1089-3539.