Forparafaçafim

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Rita de Cássia Cazu Soldi

AUTETESE: AUTOMAÇÃO DA EXECUÇÃO DE TESTES DE SOFTWARE EMBARCADO

Florianópolis

2014

Rita de Cássia Cazu Soldi

AUTETESE: AUTOMAÇÃO DA EXECUÇÃO DE TESTES DE SOFTWARE EMBARCADO

Dissertação submetida ao Programa de Pós-Graduação em Ciências da Computação para a obtenção do Grau de Mestre em Ciências da Computação.
Orientador: Prof. Dr. Antônio Augusto Medeiros Fröhlich

Florianópolis

2014

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Rita de Cássia Cazu Soldi

AUTETESE: AUTOMAÇÃO DA EXECUÇÃO DE TESTES DE SOFTWARE EMBARCADO

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciências da Computação", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciências da Computação.

Florianópolis, 10 de dezembro 2014.

Prof. Dr. Ronaldo dos Santos Mello

Prof. Dr. Antônio Augusto Medeiros Fröhlich
Orientador

Banca Examinadora:

Prof. Dr. ...

Prof. Dr. ...

AGRADECIMENTOS

Agradeço à Universidade Federal de Santa Catarina (UFSC) e ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) pelo apoio institucional para a realização deste trabalhos. Obrigada aos servidores e professores, em especial à Katiana de Castro Silva e ao Prof. Dr. Ronaldo dos Santos Mello, que me trataram com simpatia, respeito e solidariedade. Agradeço também:

Ao meu orientador, Prof. Dr. Antônio Augusto Medeiros Fröhlich (Guto). Muito obrigada pela oportunidade de fazer parte do Laboratório de Integração de Software e Hardware (LISHA) e por me guiar no desenvolvimento deste trabalho. As nossas discussões, trocas de e-mails e conhecimento foram imprescindíveis para o meu crescimento profissional e pessoal.

Aos meus colegas e amigos do LISHA, em especial o Mateus Krepsky Ludwich e o João Gabriel Reis. O apoio e a preocupação de vocês na realização deste trabalho foi fundamental, bem como as nossas discussões técnicas e científicas durante todos estes anos.

À minha família: meus amados pais Célia Regina Cazu Soldi e Luis Carlos Soldi, minha irmãzinha Luiza Helena Cazu Soldi e o meu companheiro para todas as horas, Marcelo Ribeiro Xavier da Silva. Muito obrigada pelo amor incondicional, paciência infinita e incansável dedicação ao meu sucesso e à minha felicidade durante todos os momentos de nossas vidas.

How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?

Arthur Conan Doyle

RESUMO

O número de sistemas embarcados já supera a quantidade de habitantes do nosso planeta e este número continua crescendo em ritmo acelerado. Ademais, o projeto de *hardware* e *software* está cada vez mais sofisticado e com requisitos mais rígidos para atender o exigente mercado.

O aumento da complexidade do sistema dificulta a validação e verificação dos sistemas embarcados. As consequências desta sofisticação afetam muito o desenvolvimento de *software* embarcado que. Mesmo representando uma parcela minoritária do sistema embarcado, o *software* tornou-se responsável por cerca de 80% dos erros encontrados nos sistemas.

Teste e depuração de *software* não é trivial, uma vez que é necessária a inspeção de todo o código fonte para se certificar de que o comportamento não difere das expectativas. Realizar essas atividades em sistemas embarcados é ainda mais desafiador, uma vez que os desenvolvedores precisam descobrir como otimizar o uso dos recursos, pois o teste em si tende a competir com o aplicativo sob teste pelos escassos recursos do sistema.

Esta dissertação apresenta uma maneira de ajudar os desenvolvedores no processo de testar e depurar sistemas embarcados. Ela apresenta Automação da execução de testes de *software* embarcado (AUTETESE), um ambiente que executa os casos de teste e emula as possíveis configurações do sistema, a fim de tentar encontrar erros na aplicação. Uma vez detectado um comportamento não especificado, o ambiente automaticamente executa a compilação, a depuração e a emulação de acordo com um arquivo de especificação.

O AUTETESE é avaliado de maneira quantitativa para os critérios de tentativas realizadas *versus* a configuração de granularidade, o consumo de tempo para realizar o teste e depuração e o consumo de memória para suportar a execução de testes. Adicionalmente, o ambiente é avaliado de maneira qualitativa em um comparativo com ferramentas e técnicas correlatas. Os resultados mostraram que a estratégia proposta resultou em um ambiente flexível e com grande cobertura dos desafios propostos para atingir a automação de testes de *software*.

Palavras-chave: Execução de testes automática, testes de *software* embarcado, depuração de *software* embarcado, dependabilidade

ABSTRACT

The number of embedded systems already exceed the number of inhabitants of this planet and this number continues to grow. Moreover, the design of hardware and software are increasingly sophisticated and more stringent requirements to meet the demanding market.

Increasing system complexity hinders validation and verification of embedded systems. The consequences of this sophistication greatly affect the development of the embedded software development. Even representing a minority of the embedded system, the software became responsible for about 80% of the errors found in the systems.

Software testing and debugging is not trivial, once it needs a inspection of the entire source code to make sure that the behavior does not differ from expectations. Perform these activities in embedded systems is even more challenging, since developers need to figure out how to optimize the use of resources because the test itself tends to compete with the application under test for scarce system resources.

This work presents a way to help the developers in the process of testing and debugging embedded systems. It features the AUTETESE, an environment that runs the test cases and emulates the possible system settings in order to try to find errors in the application. Once detected an unspecified performance, the environment automatically performs compilation, emulation and debugging accordingly to the specification file.

AUTETESE is evaluated quantitatively for the criteria of attempts *versus* granularity configuration, the time consumption for testing and debugging, and memory consumption to support the execution of tests. In addition, the environment is evaluated in a qualitative manner in comparison with related tools and techniques. Results show that the proposed strategy resulted in a flexible environment with high coverage to meet challenges posed by automation of software testing.

Keywords: Automatic testing execution, embedded software testing, embedded system debugging, dependability

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Sistemas embarcados podem ser apresentados como uma combinação de *hardware* e *software* concebida para realizar uma tarefa específica. Estes sistemas estão amplamente acoplados a inúmeros dispositivos e suas funcionalidades estão cada vez mais intrínsecas no cotidiano das pessoas (CARRO; WAGNER, 2003). 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).

Como em qualquer sistema computacional, as soluções embarcadas também basearam-se na premissa de que cada componente utilizado está efetuando corretamente as suas atividades e que a integração entre os mesmos não se desvia do comportamento esperado. Caso contrário, o sistema pode apresentar erros.

É comum que a consequência de um erro resulte em perdas financeiras, como por exemplo a perda de uma determinada quota de mercado, corrupção de dados do cliente, etc (TASSEY, 2002). Os danos materiais são inconvenientes, mas colocar em perigo a vida humana é um risco inaceitável. Infelizmente não faltam exemplos de erros em sistemas que resultaram em risco de morte, como a falha do sistema de defesa contra mísseis (Patriot), a ruptura do oleoduto de Bellingham - Washington, ou nos casos de overdose de radiação no tratamento de câncer ocasionadas pelo Therac-25 (ZHIVICH; CUNNINGHAM, 2009). Também existem vários exemplos relacionados à erros em sistemas embarcados, como por exemplo as tragédias aeroespaciais de Ariane 501, *The Mars Climate Orbiter* (MCO), *The Mars Polar Lander* (MPL), entre outros (LEVESON, 2009).

Grande parte destas catásrofes possui um relatório descrevendo como ocorreu o acidente, os quais indicaram que a causa mais comum das falhas continua sendo subestimar a complexidade do sistema e superestimar a eficácia dos testes (LEVESON, 2009, 2004; **??**). Mesmo depois de várias perdas relacionadas a erros em sistemas computacionais, ainda há muita complacência quando um sistema desvia do comportamento esperado, o que acaba subestimando a consequência deste erro. Uma boa prática garantir que os componentes vão agir de acordo com o requisitado é verificar de maneira pragmática cada detalhe do sistema e sem subestimar a sua complexidade.

Durante o planejamento e desenvolvimento dos testes, é necessário lembrar que o teste em si não é um comportamento do sistema e nunca deve interferir no fluxo de atividades que está sendo testado. Quando este teste tem como alvo um sistema embarcado é importante ressaltar que tanto a complexi-

dade do sistema quanto a do teste são maiores, pois estes sistemas 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 desta diversidade de opções. A escolha e integração destas ferramentas é uma etapa importante na construção de sistemas embarcados e deve ser feita de maneira criteriosa, uma vez que pode simplificar todo o processo de desenvolvimento.

1.1 MOTIVAÇÃO

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 instruções *assembly* e incoporou novas funcionalidades, que antes era oferecidas apenas pelas linguagens de alto nível.

Mesmo com este acúmulo de atividades, o *software* embarcado ainda representa uma parcela minoritária do sistema (EBERT; JONES, 2009). Esta crescene complexidade não é acompanhada pela geração de novas ferramentas para o auxílio ao desenvolvimento do *software* embarcado.

Adicionalmente, constatou-se que grande parte das ferramentas disponíveis para automação da execução de testes não são de fácil compreensão/operação e poucas 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 como sugestões 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 (TORRI et al., 2010). Desta forma, a motivação deste trabalho está em diminuir a dificuldade para execução de testes em sistemas embarcados, para que seja possível aproveitar melhor as oportunidades que são oferecidas pela indústria e tecnologia.

1.2 OBJETIVO

O principal objetivo deste trabalho é identificar o estado da arte do processo de teste e depuração 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*.

1.2.1 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*.

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 da área de teste e depuração de sistemas embarcados.
- Desenvolver uma arquitetura que realize a automação da execução de testes de maneira simples, sem que seja necessário configurar cada teste a ser executado.
- Especificar de um ambiente capaz de integrar a execução automática dos testes e a depuração de um *software*.
- Realizar uma avaliação qualitativa e quantitativa da arquitetura proposta, comparando o presente trabalho com os trabalhos relacionados.
- Apresentar o trabalho desenvolvido 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 técnica/científica.

1.3 ORGANIZAÇÃO DO TEXTO

- **Capítulo 2**: apresenta os conceitos fundamentais da área, compondo uma base teórica para auxiliar no desenvolvimento das ideias e na assimilação do ambiente proposto neste trabalho.
- **Capítulo 3**: possui os trabalhos que representam o estado da arte da literatura e que servem de inspiração para o desenvolvimento desta dissertação. Todos os trabalhos aqui apresentados se relacionam diretamente com o tema abordado.

2 CONCEITOS BÁSICOS

A evolução tecnológica e o crescente uso de sistemas computacionais culminou na produção de sistemas mais sofisticados e complexos, que exigem um projeto sistemático e rigoroso. Porém, o desenvolvimento de sistemas, por mais cauteloso que seja, está sujeito a erros. O teste de *software* é aplicado para tentar eliminar os erros inseridos durante a elaboração de um sistema (PRESSMAN, 2011).

2.1 TESTABILIDADE DE UM SISTEMA

A testabilidade de um sistema é o grau de facilidade para se estabelecer critérios de teste e para executar os testes, a fim de determinar se os critérios foram atendidos (IEEE..., 1990). O conceito de testabilidade de um componente de *software* foi primordialmente introduzido por Freedman (FREEDMAN, 1991) como uma combinação de controlabilidade e observabilidade.

A controlabilidade e observabilidade interferem diretamente na viabilidade do projeto, da execução e da automação de testes. Além destas duas características iniciais, outras quatro foram elencadas como sugestões de como projetar um *software* para se atingir um alto grau de testabilidade (BACH, 2003; CHOWDHARY, 2009; PRESSMAN, 2011):

Controlabilidade - capacidade de controlar o processo, conduzir os testes e estados das variáveis. Um sistema é controlável quando é possível, à partir de uma determinada entrada, transferir o sistema de um estado inicial X_{t_0} para um outro estado X_{t_f} durante o intervalo finito de tempo $(t_f - t_0)$.

Observabilidade - examinar os registros dos estados do sistema, de visualizar todos os elementos que afetam os estados do sistema e de identificar saídas incorretas. Um sistema é observável quando uma determinada condição inicial do estado X_{t_0} pode ter seu comportamento relatado durante o intervalo finito de tempo t, onde $t_0 \leq t \geq t_f$.

Disponibilidade - facilidade de acesso ao sistema para poder aplicar os testes. Em um sistema disponível deve fornecer acesso ao código fonte e, principalmente, não conter defeitos impeditivos para a execução dos testes.

Operabilidade - execução do sistema deve ocorrer de maneira simples e clara. Um sistema operável é desenvolvido para atender os requisi-

tos com a maior simplicidade possível, à partir de componentes coesas e com baixo acoplamento.

Estabilidade - resiliência à mudanças. Um sistema possui estabilidade quando as alterações não são frequentes, e as manutenções necessárias realizadas de maneira controlada e sem invalidar os testes já existentes.

Compreensível - facilidade de compreensão do sistema. Existem algumas características que podem deixar um sistema mais fácil de ser assimilado. Alguns exemplos são: a escolha de tecnologias adequadas para auxiliar no desenvolvimento do sistema, documentação acessível e detalhada, o uso de padrões de projeto e desenvolvimento, etc.

2.2 TESTE DE *SOFTWARE*

A primeira conferência com foco em teste de *software* ocorreu há 42 anos, em junho de 1972 (GELPERIN; HETZEL, 1988). Desde então há um esforço para definir melhor os conceitos relacionados à área e também para chegar-se a um consenso sobre pontos chave como: nomenclatura, documentação, especificação e execução de testes.

Esta seção traz os principais pontos de discussão e definições na área de teste de *software*.

2.2.1 Defeito, erro e falha

Os termos defeito, erro e falha estão diretamente relacionados ao teste de *software* e suas designações foram estabelecidas pela *Institute of Electrical and Electronics Engineers* (IEEE) (IEEE..., 2010). Estes termos são semelhantes aos utilizados para designar erros de programação, entretanto, as expressões não são sinônimas entre si e são usadas para expressar conceitos distintos (KOSCIANSKI; SOARES, 2007). Nos escopo deste trabalho os termos ficam estabelecidos - e ilustrados pela Figura 1 - como:

- **Defeito** é caracterizado como um passo, processo, ou definições que são realizadas por um indivíduo de maneira equivocada. O defeito é a causa de um erro, e normalmente ocorre em um nível base, no universo físico, ao abstrair informações, definir um requisito, ao inserir uma linha de código fonte.
- **Erro** é uma manifestação concreta de um defeito num artefato de *software*. Eles ocorrem no universo da informação, e se sobressaem du-

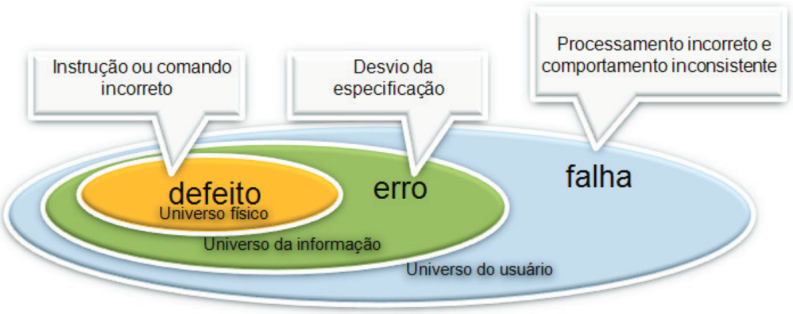

Figura 1 – Defeito *vs* erro *vs* falha (NETO, 2012)

rante a execução do *software*. Caracterizam-se pela divergência entre os resultados obtidos e os esperados de um *software*, pois geram um resultado inesperado na execução de um *software*.

- **Falha** é um desvio da especificação e surge quando existe um processamento incorreto que leva a um comportamento diferente do esperado pelo usuário. Uma falha pode ser considerada com uma deficiência do *software* em cumprir determinados requisitos funcionais devido à ocorrência de um ou mais erros.

2.2.2 Objetivos do teste de *software*

A definição com maior aceitação apresenta o teste de *software* como o processo de análise de um item de *software* para detectar as diferenças entre as condições existentes e exigidas (isto é, os erros), e avaliar as características do item de software (ANSI/IEEE, 1986). Sendo assim, esta atividade é amplamente utilizada para assegurar que o *software* contempla as funcionalidades especificadas de maneira correta.

Segundo Myers (MYERS; SANDLER, 2004), os objetivos do teste de *software* podem ser expressos a partir da observação de três regras:

- A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro.
- Um bom caso de teste é aquele que apresenta uma elevada probabilidade de revelar um erro ainda não descoberto.
- Um teste bem sucedido é aquele que revela um erro ainda não descoberto.

Estas três regras ressaltam o objetivo do teste de *software*, que visa salientar a presença dos erros e não sua ausência. Isto significa que o sucesso de um teste em encontrar um desvio de comportamento em um determinado *software* não garante que todas as possíveis falhas tenham sido encontradas. O teste que não encontra erros é meramente inconclusivo, pois não se sabe se há ou não desvio no comportamento do *software* analisado.

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 de maneira eficaz, deve-se fazer revisões técnicas 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 a alguma estratégia de teste.

Testar um *software* é essencial para revelar o número máximo de erros a partir do menor esforço, ou seja, quanto antes forem detectadas divergências entre o que foi requisitado e o que foi entregue menor será o desperdício de tempo e esforço de retrabalho para adequar o *software*.

2.2.3 Técnicas de teste de *software*

A escolha da abordagem e da seleção dos casos de testes de *software* podem ser categorizados conforme a fonte de informações em que se baseiam (PEZZÈ; YOUNG, 2008). Tomando por base estes critérios, os diferentes tipos de testes de *software* podem ser classificados em dois grandes grupos: o dos testes funcionais e o dos testes estruturais (FRANZEN; BELLINI, 2005).

2.2.3.1 Teste funcional

O teste funcional, também conhecido como teste de caixa preta ou teste comportamental, é baseado em especificações do *software* e suas entradas são selecionadas à partir de uma análise das funcionalidades do sistema.

Neste tipo de teste a estrutura de controle do sistema é propositalmente desconsiderada e concentra-se os esforços do teste no domínio da informação. Sendo assim, os casos de teste devem ser derivados dos requisitos funcionais do sistema.

Os dados de entrada para os casos de teste podem ser obtidos de maneira aleatória, através de especificações ou através das funções do *software*, gerando um conjunto de equivalências entre os valores de entrada e os valores de saída. Já a identificação dos erros é feita inserindo um determinado valor de entrada e contrastando o resultado esperado com o resultado obtido.

Existem alguns métodos de teste funcional e dentre os principais estão:

Método de teste com base em grafo - consiste em abstrair o sistema como um grafo, onde os nodos são as possíveis entradas e saídas do sistema, e suas conexões expressam algum tipo de ação ou requisito para deslocar-se de um nodo para o outro. Este grafo será organizado em forma de tabelas de causa-efeito e utilizado como base para a geração dos casos de teste.

Particionamento de equivalência - classifica as entradas do sistema em grupos que tendem a utilizar uma determinada função do *software* e a partir deste grupo são derivados os casos de teste. Estes casos de teste são elaborados para um representante de cada classe ao invés de realizar testes exaustivos com todos os membros. O foco é verificar o máximo de atributos da classe de equivalência ao mesmo tempo.

Análise do valor limite - investiga a capacidade do sistema em manipular os dados dentro de seu limite. Este método concentra-se em injetar valores que estão nas fronteiras do sistema, onde os erros ocorrem com maior frequência. Este método é bastante utilizado em conjunto com o particionamento de equivalência, extrapolando os valores limites de cada agrupamento.

Matriz ortogonal - tem como objetivo a construção de casos de teste utilizando-se de uma visualização geométrica do domínio de entradas de um *software*. Com a matriz, as entradas ficam dispersas de maneira uniforme por todo o domínio do teste. Este modelo é utilizado quando a quantidade de dados de entradas, apesar de limitada, é muito grande para realizar testes exaustivos.

Para utilizar-se desta técnica, o ideal seria conter casos de teste que contemplem todas as entradas permitidas para o sistema, entretanto, na maioria dos sistemas isto pode não ser possível (MYERS; SANDLER, 2004).

O teste funcional possui êxito em investigar alguns tipos de erros mas, como o modo de identificação do erro desconsidera o estado interno do sistema, os casos de teste podem não ser suficientes para identificar certos riscos num projeto de *software*. Este tipo de técnica atinge erros nas seguintes categorias (PRESSMAN, 2011):

- Funções incorretas ou defectivas
- Erros de interface
- Erros em estrutura de dados
- Deficiência no acesso em bases de dados ou serviços externos
- Comportamento anômalo do sistema
- Problemas com desempenho
- Erros na inicialização ou término do *software*

2.2.3.2 Teste estrutural

O teste estrutural (ou teste de caixa branca) possui como estratégia a derivação dos dados de teste a partir de uma análise da lógica do *software* (MYERS; SANDLER, 2004). Ou seja, esta técnica baseia-se na estrutura interna do sistema, analisando os dados do *software*, o estado interno dos módulos e o fluxo de controle.

O projeto de casos de teste é dependente da implementação do sistema e utiliza a estrutura de controle descritas como parte do projeto no nível de componentes. Com a técnica de testes estruturais os casos de testes focam nos seguintes aspectos (PRESSMAN, 2011):

- Garantir que todos os caminhos independentes do sistema sejam exercitados pelo menos uma vez
- Exercitar todas as decisões lógicas, investigando os estados verdadeiro e falso
- Executar todos os ciclos dentro de seus limites e nas fronteiras operacionais

- Analisar as estruturas de dados internos, a fim de assegurar a validade

Hetzel indica o teste caixa branca como ”teste no pequeno”, pois são tipicamente desenvolvidos para pequenos componentes do sistema (HETZEL, 1988; PRESSMAN, 2011). Isto ocorre devido ao nível de detalhismo necessário para desenvolver os testes estruturais. Dentre os principais métodos utilizados para gerar casos de teste estão:

Caminho básico - projeto de casos de teste para considerar que todas as instruções do *software* são executadas pelo menos uma vez. O primeiro passo deste método é desenhar um grafo de fluxo representando o *software*, onde os nós representam os processos e as arestas são o fluxo de controle. A partir deste grafo determina-se a complexidade ciclomática e os conjuntos de caminhos básicos, linearmente independentes. Os casos de teste são derivados do conjunto de caminhos básicos de acordo com um critério de cobertura que irá assegurar que cada declaração foi exercitada, mas isso não significa que todos os resultados da decisão foram executados (FARRELL-VINAY, 2008).

Estrutura de controle - projeto de casos de teste derivado de características do fluxo de execução do *software* como, por exemplo, laços, condições, comandos e desvios. Existem algumas variações de testes de estrutura de controle: (i) testes de condição, que verifica as condições lógicas contidas no código fonte do *software*; (ii) teste de fluxo de dados, que elege os caminhos de teste de acordo com o uso de variáveis e dados internos; (iii) teste de laços, que valida a construção dos laços de repetição.

Independente da técnica utilizada, os testes devem ser adotados desde o início do projeto do *software*, considerando a definição dos requisitos a serem testados, a política de testes, os 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.

2.3 DEPURAÇÃO DE *SOFTWARE* EMBARCADO

A depuração é o processo de diagnosticar erros em sistemas e determinar a melhor forma de corrigi-los (PRESSMAN, 2011). Em geral, o ponto

de partida para a depuração de um sistema é a ocorrência de algum erro, portanto, é muito comum que a depuração ocorra como consequência de um teste bem sucedido, isto é, de um teste que encontrou erros.

Erros podem ser encontrados por qualquer pessoa e em qualquer etapa do ciclo de vida do *software*. Isto implica que tanto o formato do relatório de um resultado inesperado, quanto as características dos erros 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 de *software* embarcado 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. No trabalho de Hopkins e McDonald-Maier (HOPKINS; MCDONALD-MAIER, 2006) são definidas as seguintes diretrizes para suporte a uma depuração eficaz:

- O suporte à depuração não deve interferir no comportamento do sistema.
- Deve existir uma infraestrutura para observar o estado do sistema e de possíveis pontos críticos.
- Garantia de acesso para controlar o estado interno do sistema e de seus recursos (incluindo periféricos).
- O sistema deve ser minimamente impactado, principalmente quando são necessárias mudanças no *hardware*.

O apoio à depuração de *software* normalmente ocorre a partir do monitoramento do *software* compilado e instrumentado. A instrumentação pode ser aplicada e suportada de várias maneiras, dentre as mais frequentes estão a execução do *software* em modo de depuração, o uso de tratadores de interrupções e o uso de instruções de suporte à depuração.

2.3.1 Imprimir dados em dispositivos de saída

Imprimir dados em dispositivos de saída pode ser considerada como uma técnica de instrumentação e depuração, principalmente quando é utilizada com o intuiuto de exibir/registrar variáveis, coletar informações sobre o estado do sistema ou até o progresso da execução do *software* através do código do programa.

Esta técnica de depuração é primitiva e intrusiva, pois normalmente está associada à mudanças no código fonte e recompilação do *software*. Dependendo de como utilizada, pode-se retardar a execução do sistema, modificando inclusive requisitos do próprio *Software Under Test* (SUT). Portanto, para depurar utilizando a impressão de dados em dispositivos de saída, é necessário que o erro possa ser reproduzido e que ele não se altere com as impressões.

Esta maneira de depuração pode introduzir efeitos colaterais que mascarem o verdadeiro problema. Isto torna ainda mais difícil a identificação de erros relacionados à operações temporais, de paralelismo e de alocação de recursos.

2.3.2 Emulador em circuito

In-circuit emulator (ICE) é um *hardware* utilizado em depuração de projeto de sistemas embarcados, fornecendo recursos e conectividade que ajudam a superar as limitações de teste e depuração tanto do *software* quanto do *hardware*.

Este equipamento é específico para cada microprocessador, por isso permite controle total do processador e viabiliza a observação das operações do sistema de maneira consistente e eficaz. A informação coletada por ele fica disponível em tempo de execução e, a princípio, o equipamento extra não exerce nenhum impacto sobre o comportamento do SUT.

As placas ICE conseguem integrar o controle de execução do microprocessador, acesso à memória e monitoração em tempo real. Por isso é uma estratégia muito utilizada no início do desenvolvimento dos sistemas embarcados. Entretanto, o aumento da complexidade do sistema pode exigir adaptações no projeto de integração das ICEs ao sistema. No caso de sistemas complexos o esforço e o custo para realizar tal adaptação pode ser tão grande que tornam a depuração proibitiva (HOPKINS; MCDONALD-MAIER, 2006).

2.3.3 Depuradores

Depuradores são ferramentas que auxiliam no monitoramento da execução de um programa, incluindo 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 considerados na construção de um ambiente de depuração estável.

No método local o programa é executado na mesma máquina que o depurador. Isto implica em um processo com latência menor e com grande influência entre ambos. Por exemplo, se um processo provoca um *crash* no sistema, o depurador perde grande parte das informações de deputação, pois pertence ao mesmo ambiente que sofreu a parada e 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 a aplicação sob depuração e o depurador encontram-se em máquinas separadas. Do ponto de vista do ambiente de depuração, a depuração remota é semelhante a uma depuração local com duas telas conectadas em um único sistema.

2.4 PROJETO DE SISTEMAS ORIENTADO A APLICAÇÃO

A metodologia de projeto de sistemas orientado a aplicação, a *Application-Driven Embedded System Design* (ADESD), 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 desde o início do projeto. Desta forma, 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:

Famílias de abstrações independentes de cenário

Durante a decomposição de domínio as abstrações identificadas são agrupadas em famílias de abstrações e modeladas como membros desta família. 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.

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.

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

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

A Figura 2 mostra uma visão geral da ADESD, 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.

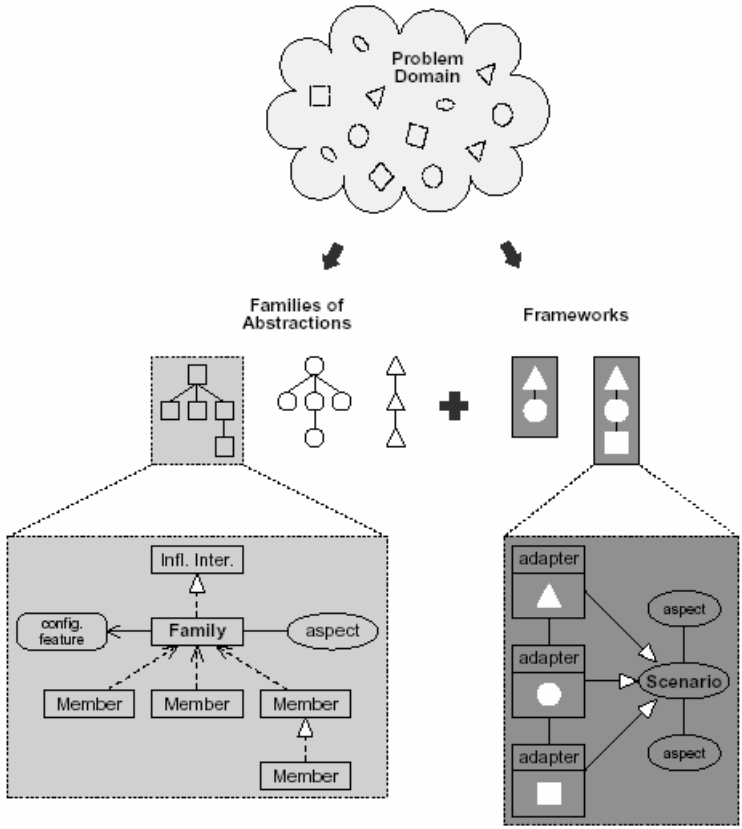

Figura 2 – Visão geral da metodologia de projeto de sistemas orientado à aplicação (FRÖHLICH, 2001)

3 TRABALHOS RELACIONADOS

Desde a formulação da área de testes de *software*, muitos estudos surgiram com as mais variadas soluções para a automação deste processo. Apesar de existir uma vasta literatura de apoio, a maior parte dela relaciona-se à sistemas cujo o propósito geral e, portanto, apenas alguns conceitos podem ser aplicados à sistemas embarcados.

Os trabalhos que focam na automação de teste de *software* embarcado procuram soluções em que o teste não prejudique a execução do SUT e ainda consiga contornar as restrições inerentes ao próprio sistema embarcado.

3.1 JUSTITIA

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

Justitia possui a hipótese de que a interface é um critério essencial para um teste de *software* embarcado, e para defendê-la utiliza uma solução que une as seguintes técnicas:

- **Técnica de teste de interface**, na qual os casos de teste são gerados 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*. Quando teste é executado são extraídas as informações de depuração, tais como símbolos, relações entre os componentes, informações do *driver* de dispositivo, etc.

- **Técnica de emulação dos casos de teste**, uma combinação entre o monitoramento e a depuração do sistema. A ideia é definir *breakpoints* para os casos de teste da interface e monitorar as variáveis de interface para definir o sucesso ou falha do teste. As variáveis de interface desempenham o papel de indicador para o funcionamento do sistema embarcado,verificando se alguma camada está instável e encapsulando a influência do *hardware* sobre falhas em *software* embarcado

A solução foi analisada sob três pontos de vista: a densidade da interface, a complexidade do *software* e a relação interface *versus* falhas. Os pontos de análise surgiram de uma premissa que aponta o forte acoplamento

entre *hardware* e *software* como a razão para aumento de dificuldades no teste de *software* embarcado.

Sendo assim, *Justitia* apresenta uma análise da densidade da interface, na qual é estabelecida uma relação entre o grau de acoplamento *software* e sua testabilidade. Já a complexidade do *software* é atualmente uma das principais métricas para a previsão de falhas no sistema e no caso de *Justitia* pode ser extrapolada para estimar a probabilidade de falhas na interface. O último ponto de vista aponta a análise da relação entre interface *versus* falhas como base para os resultados encontrados pela ferramenta.

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 falhas. A solução foi novamente corroborada por uma versão mais recente (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.

Conclusão

Este trabalho ganhou destaque ao apontar que a maior parte das falhas nos testes de *software* embarcados são oriundos da integração de componentes heterogêneos e ao propor que a interface seja um ponto de intersecção entre os testes de *software* e *hardware*.

Outro diferencial encontra-se na infraestrutura de execução dos testes baseada em emulação, na qual tanto o resultado da execução do SUT quanto o monitoramento da tabela de símbolos são utilizados para determinar o sucesso ou falha do teste.

3.2 ATEMES

ATEMES (KOONG et al., 2012) é uma ferramenta para automação de teste para sistemas embarcados *multicore*. 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** - módulo de pré-processamento - cujas atribuições consistem em a análise de código fonte, geração automática de casos de teste, geração automática dos dados de entrada de teste e instrumentação do código fonte.
- **HSATM** - módulo de teste automático do lado *host* - é responsável por

gerar automaticamente 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** - módulo de teste automático da plataforma de *hardware* alvo - tem como principal função a execução da imagem recebida do $HSATM$, além de monitorar o andamento dos testes e enviar os dados desta execução.

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

A partir destes componentes é possível executar os testes em uma plataforma alvo a partir de uma estação de trabalho remota, diminuindo restrições quanto ao uso de recursos dos sistemas embarcados. Para tanto, o *software* passa por uma compilação cruzada no lado *host* (estação de trabalho) e é enviado automaticamente para a plataforma alvo onde é executado.

As atividades são gravadas em um registro, dentre estes registros 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 registro pode ser passado para o lado *host* em tempo de execução, onde pode ser armazenado, processado e até apresentado de maneira visual através de uma interface.

Conclusão

Ao explorar sistematicamente a automação de testes de *software* embarcado e componentizar cada operação necessária, $ATEMES$ apresenta uma solução de grande desempenho para sistemas multiprocessados. A ferramenta é tão robusta que é possível inclusive automatizar a geração de casos de testes à partir da análise do código fonte.

Conforme será visto no Capítulo **??**, a abordagem de execução automática de testes de $ATEMES$ é similar ao deste presente trabalho. Ambos utilizam depuração cruzada para compilar o código fonte para uma determinada arquitetura de *hardware* e monitoram a execução desta imagem à partir de uma conexão remota. O registro dos dados de execução dos testes e a apresentação de um relatório para o usuário também é uma semelhança notória.

3.3 AUTOMAÇÃO DA EXECUÇÃO DE TESTES BASEADA NA INTERFACE DO SISTEMA ALVO

Este trabalho apresenta uma abordagem o testes de sistemas embarcados heterogêneos e baseada nas características do SUT (KARMORE; MABAJAN, 2013).

Diferente da escolha tradicional, que normalmente é feita a partir de uma mistura de vários modelos para tentar alcançar o maior número possível de arquiteturas, a *Target Based embedded system Testing* (TBT) reforça que cada sistema embarcado possui características únicas e que devem ser consideradas na escolha do modelo de teste que será aplicado ao sistema.

A *framework* criada utiliza-se de um módulo *Discovery Interface Device* (DID) que fica conectado diretamente com o sistema embarcado. Este módulo é o responsável por detectar o sistema, descobrir suas características (*drivers* e interfaces) e apontar os testes que melhor se encaixam ao perfil do sistema conectado.

Entretanto, para que seja possível a extração das informações do sistema alvo e realizar a execução automática dos testes o SUT deve ter o projeto de testes baseado em interfaces, ou seja, cada módulo quem compõe o sistema deve implementar uma determinada interface de testes.

Com estas informações, o mecanismo de teste da *framework* consegue sugerir e executar três tipos de teste. Os testes gerais são sugeridos para sistemas que compõem o sistema embarcado como, por exemplo, capacitores, resistores, *leds*, sensores, etc. Os testes internos verificam cada componente em separado e podem ser realizados com a ajuda de ferramentas que geram algoritmos para sistemas embarcados. Os testes externos verificam o sistema como um todo, contemplando também a integração do sistema.

Em uma nova versão da abordagem (PRIYA; MANI; DIVYA, 2014) o mecanismo de teste foi adaptado para considerar um oráculo, composto por uma rede neural de 2 camadas com retroalimentação. O treinamento desta rede neural ocorre a partir de execuções corretas do sistema embarcado até que o oráculo tenha um modelo de simulação do *software* com a precisão desejada.

Quando novas versões do sistema original forem lançadas, o *software* é novamente executado e sua saída é incorporado através da retroalimentação. O resultado obtido pelo ensaio deve ser comparado com a saída do oráculo para determinar se está correto ou não.

Conclusão

O destaque desta abordagem está na escolha de técnicas para teste de sistemas heterogêneos. A *framework* apresenta uma característica que

pode facilitar muito a automação de testes para sistemas embarcados, pois desenvolver testes específicos para um sistema embarcado e que ainda sejam independentes do ambiente de implantação (*deploy*) não é uma tarefa trivial.

A adaptação do mecanismo de testes para o uso de um oráculo composto por uma rede neural é com certeza outro diferencial, tornando possível que a *framework* aprenda com os próprios resultados e consiga generalizar ou especializar uma determinada configuração de testes.

No capítulo **??** será possível verificar uma grande semelhança no processo de avaliação dos testes para determinar seu sucesso/falha, principalmente quando utilizada a estratégia de comparação entre os resultados das execuções realizadas.

3.4 PARTIÇÃO DO *SOFTWARE*

O conceito de partição do *software* surgiu da afirmação de que quando as pessoas estão depurando um *software*, o processo mais comum é fazer abstrações mentais correspondentes a partes do todo. Desta forma desenvolve-se a hipótese de integrar particionadores de código fonte nos ambientes de depuração (WEISER, 1981).

Após a definição de Weiser, a partição foi utilizada em diversas áreas da computação, como paralelização, ajuste de níveis de compilação, engenharia reversa, manutenção de *software*, testes, entre outros. Na área de depuração, focou-se na partição sistemática do *software*, investindo na remoção de estados ou caminhos que são irrelevantes para alcançar o objetivo selecionado, o que no caso da depuração é encontrar um erro (XU et al., 2005).

Cada trabalho apresenta um critério diferente para particionar, mas todos possuem o conceito de fatia (*slice*) como um subconjunto de predicados do programa que afetam os valores computados, de acordo com o critério de controle. Uma distinção importante para particionar é o tipo de fatia que será realizada: estática, dinâmica ou uma variação delas (SASIREKHA; ROBERT; HEMALATHA, 2011).

A partição estática é o tipo mais rápido e, em contrapartida, 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.

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* complexo e de alto grau de acoplamento.

Além dos tipos base de partição, também existem modelos que são variações de um dos dois principais ou até gerados pela união de modelos (VENKATESH, 1991; BINKLEY et al., 2005; SRIDHARAN; FINK; BODIK, 2007; CHEBARO et al., 2012), proporcionando uma maior configurabilidade de depuração e atendendo melhor às características do SUT.

Conclusão

A automação da partição do código deriva de uma abstração tradicionalmente utilizada por seres humanos durante a depuração. Desde que foi computacionalmente implementado, surgiram inúmeras variações no critério de partição, tamanho da fatia o programa e alternativas para o cálculo das fatias. A principal razão desta diversidade está na peculiaridade de cada *software*, onde cada característica requer uma maneira diferenciada de realizar a partição.

Independente da abordagem selecionada, a partição do código é uma técnica versátil e muito utilizada, principalmente porque necessita apenas de uma execução com falhas para ser capaz de simplificar o *software* em *slices*, cujo grupo de entradas a serem examinadas é bem reduzido. Poder selecionar pequenos componentes do código fonte para seres testadas e depuradas em partes é uma das principais características da ferramenta apresentada no capítulo **??**.

3.5 DEPURAÇÃO ESTATÍSTICA

Trabalhos baseados em depuração estatística utilizam-se de 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.

O primeiro passo deve ser a instrumentação do *software* para coletar dados sobre os valores de determinados tipos de predicados em pontos específicos da execução do *software*. Existem três categorias de predicados rastreados:

- **Ramificação** - para cada condicional encontrado são gerados dois pre-

dicados: um que indica o se o caminho escolhido foi o da condição verdadeira e outro que indica a escolha do caminho caso a condição seja falsa.

- **Retorno de funções** - cada retorno de função com um valor escalar são gerados seis tipos de predicados para rastrear o valor: > 0, ≥ 0, $= 0$, < 0, ≤ 0, $\neq 0$.

- **Atribuição** - em toda atribuição escalar também são gerados seis tipos de predicados para comparar os valores: $>$, $\geq$, $=$, $<$, $\leq$, $\neq$.

Essas informações são armazenadas em relatórios como um vetor de *bits*, com dois *bits* para cada predicado - um para o resultado observado e outro para o resultado verdadeiro e um bit final que representa o sucesso ou falha da execução. Em sequência, são atribuídas pontuações numéricas para identificar qual foi o predicado que melhor expressa o conjunto de predicados.

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 identificação aproximada de qual parte do sistema gerou o erro.

3.5.1 Depuração estatística baseada em amostras

É uma das primeiras abordagens que propõe recolher os dados estatísticos através de declarações inseridas no código e, em tempo de execução, consegue coletar dicas 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 probabilidade 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.

3.5.2 Depuração estatística baseada em predicados

Este modelo 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).

A manipulação de predicados espelhados pelo código fonte não é um trabalho trivial, e isso faz com que muitos modelos não consigam selecionar padrões de erros úteis. Para mitigar este problema a depuração estatística baseada em predicados utilizada técnicas semelhantes às dos algoritmos *bi-clusters*, que permite a extração de dados bidirecionais simultaneamente.

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. Este número 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 redundância, pois o processo de votação só acaba quando houver 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.

Conclusão

Os trabalhos que utilizam-se de depuração estatística necessitam de um grande volume de dados para realizar uma análise confiável das informações e retornar resultados válidos. Este técnica é de difícil implementação em um sistema embarcado real por não estarem preparados para tal armazenamento. Entretanto, conforme será demonstrado nos capítulos seguintes, é possível coletar e armazenar estes dados no caso da depuração remota.

A depuração estatística baseada em amostras contribui com a discussão sobre a necessidade de ativar todos os possíveis estados do *software* em todas as execuções do sistema para que seja possível descobrir a origem do erro. Sabe-se que ao inserir declarações aleatórias no *software* é possível reduzir a quantidade de dados a serem analisados, visto que apenas uma parte é executada a cada rodada.

A aleatoriedade pode ser um aliada na depuração do sistema, mas deve-se tomar cuidado com a quantidade de predicados e valores que os

mesmos podem atingir. Para eliminar predicados pouco representativos, alternativas como *ranking* e votação foram apresentadas. Levando em conta estes aspectos, o presente trabalho possui no Capítulo **??** uma discussão sobre a configuração da do sistema, para que a aleatoriedade não prejudique o resultado desejado.

3.6 DEPURAÇÃO POR DELTA

A técnica de *Delta Debugging* (DD) estabelece uma hipótese sobre o motivo de um *software* parar de funcionar corretamente e, dependendo dos resultados da execução dos testes, esta hipótese deve ser refinada ou rejeitada (ZELLER, 1999). A ideia do DD é chegar a um mínimo local, onde todos os eventos presentes interferem diretamente no comportamento incoerente do *software*.

Assumindo-se que o erro é determinístico e que pode ser reproduzido automaticamente, é possível refinar a hipótese ao remover de forma iterativa os trechos do *software* que não colaboram na ocorrência do erro. O mesmo ocorre com a rejeição da hipótese, quando um trecho é retirado e o *software* não apresenta mais o erro.

Para automatizar a escolha, refinamento e rejeição de hipóteses, a DD trabalha com dois algoritmos:

- **Simplificação** - este algoritmo foca na simplificação da entrada que ocasionou a falha do teste. Para tanto, a entrada é analisada e tenta-se reduzir ao máximo sua complexidade, até que não seja mais possível simplificá-la sem eliminar o fracasso dos testes.

- **Isolamento** - algoritmo para encontrar uma configuração na qual a adição ou remoção de um elemento tem influência direta no resultado dos testes. Ele é ideal para encontrar os subconjuntos de um caso de falha.

A automação dos testes e da depuração procura determinar as causas do comportamento inadequado a partir da observação as diferenças (deltas) entre versões. Ela pode ser utilizada para simplificar ou isolar causas da falha e pode ser aplicado a qualquer tipo de dado que afete de alguma forma a execução do SUT.

Nesta abordagem é comum trabalhar com o teste e a depuração em conjunto, inclusive porque refazer a execução dos testes de um *software* sob novas circunstâncias é uma técnica comum de depuração e, muitas vezes, é a única maneira de provar que as novas circunstâncias realmente podem causar a falha (ZELLER; HILDEBRANDT, 2002).

3.6.1 Depuração por delta com hierarquias

A técnica genérica de DD utiliza os algoritmos de simplificação e isolamento para fornecer um subconjunto onde encontra-se a mudança que possivelmente causou o erro. Porém, em casos mais complexos o DD perde um pouco de eficácia e aponta para vários subconjuntos com falha nos testes e cujas entradas não são tão simplificadas.

O HDD - *Hierarchical Delta Debugging* (MISHERGHI; SU, 2006) surgiu para melhorar o desempenho dos algoritmos do DD, principalmente quando o código fonte analisado possui uma estrutura hierárquica e semântica, ou seja, o SUT não é apenas uma estrutura de linhas desconexas e independentes.

A ideia é aproveitar a estrutura hierárquica da entrada para reduzir o número de tentativas inválidas, pois explorando a estrutura dos dados de entrada é possível melhorar a convergência e chegar mais rápido à estrutura simplificada.

Para considera a hierarquia, o algoritmo de simplificação DD foi modificado para operar em uma árvore de sintaxe abstrata e sua implementação foi aplicada para avaliar arquivos *eXtensible Markup Language* (XML) e programas escritos em *C*. A versão XML explora diretamente a estrutura hierárquica da entrada, extraindo subárvores, folhas, atributos e até caracteres. Para a versão *C* foi necessário gerar um analisador do código fonte, essencial para manipular o *software* de acordo com a árvore de sintaxe simplificada.

3.6.2 Depuração por delta e iterativa

Para identificar o mínimo local a DD necessita de duas versões do *software*: uma falha e uma correta. Contudo, quando um novo erro é descoberto, pode ser que a versão anterior também contenha o mesmo erro e será necessário procurar em versões ainda mais antigas, até encontrar uma versão correta para aplicar o delta.

A escolha desta de versões foi abordada pelo *Iterative Delta Debugging* (IDD) (ARTHO, 2011), o qual automatizou a busca dessa versão correta através de comparações iterativas. O conjuntos de testes também são executados de forma automática para cada iteração e o algoritmo prosseguem até que uma das três condições sejam atingidas:(i) a versão que passa pelos testes é encontrada, (ii) não haja mais versões antigas disponíveis no repositório, ou (iii) foi alcançado o tempo limite da busca.

Durante a comparação, sempre que a execução do *software* não alcança o sucesso porque diverge do esperado (ex. erros de compilação, *looping* in-

finito, mudanças sintáticas), os algoritmos de DD fazem pequenos *patch* para preservar o resultado anterior. No caso de não ser possível aplicar o *patch*, o IDD considera o resultado como fracasso.

Para garantir que os algoritmos do DD não remova trechos de código que contribuem para o valor do resultado do teste, o IDD também automatizou o monitoramento de dados importantes da execução, dos arquivos de log, consumo de memória e o tempo necessário para executar o *software* tanto para ambas versões do delta.

Conclusão

A base da DD está em analisar a diferença entre duas versões do sistema e isolar/analisar as possíveis causas do comportamento inesperado automaticamente quando alguma divergência é identificada na execução da nova versão.

Se o delta for bem aplicado, a estrutura que resta aponta apenas para os elementos relevantes no fracasso do teste do *software*. Entretanto, no caso de grandes mudanças, a estrutura restante pode ser complexa e apresentar um grande número de dados para explorar.

Alguns trabalhos subsequentes concentraram em maneiras mais assertivas de reduzir o conjunto de entradas e isolar o problema. Uma delas é a *HDD*, que utiliza-se de dados referentes à hierarquia do programa para agilizar o trabalho realizado pelo algoritmo de redução. Outra alternativa interessante é o IDD, que procura o melhor delta entre várias versões disponíveis no repositório.

A comparação de versões no repositório é uma técnica interessante de depuração e pode ser aplicada inclusive para a geração de relatórios e gráficos sobre a consistência do *software*. No capítulo **??** será apresentado um algoritmo quem também se utiliza do delta para determinar sucesso/falha dos testes do SUT.

3.7 CAPTURA E REPRODUÇÃO

A ideia da técnica de captura e reprodução consiste em selecionar um subsistema do *software* como alvo do estudo, capturar todas as operações entre este subsistema e o resto do *software* e depois reexecutar as operações isoladas do subsistema (ORSO; KENNEDY, 2005).

Este tipo de depuração 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.

A ferramenta *JINSI* foi implementada para aplicar os conceitos de captura e reprodução para programas na linguagem Java (ORSO et al., 2006). Dado um sistema e um componente presente no sistema, *JINSI* grava as interações entre esse componente e o resto do sistema. Neste contexto, o componente é considerado como um conjunto de classes com uma função definida.

JINSI foi novamente abordada por Burger e Zeller (BURGER; ZELLER, 2008), adquirindo a capacidade de capturar e reproduzir as interações inter/intra componentes. Assim, todas as operações relevantes são observadas e executadas passo a passo, considerando-se todas as comunicações entre dois componentes até encontrar o *bug*.

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. Afina, interrupções podem ocorrer a qualquer momento e provocar uma ruptura no fluxo de controle e, desta forma, barrar a execução do programa na instrução atual para continuar a partir da 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.

Conclusão

Captura e Reprodução sempre foi muito utilizada para testes de interface, em que a interação do usuário com o *software* era armazenada para poder servir como caso de testes. Desta forma torna-se possível realizar testes de funcionalidade do sistema de acordo com o estímulo do usuário.

Já a depuração é um caso mais específico, pois nem todo erro pode ser reproduzido, mesmo que todas as ações sejam reexecutadas na mesma ordem. Contudo, para um erro determinístico em que se tem todas as operações que podem ter causado de um determinado erro, é possível realizar uma investigação baseada no fluxo de controle do *software*.

A ferramenta *JINSI* foi utilizada para corroborar a técnica, que além de capturar e reproduzir o *software*, agora também incorporou os algoritmos de DD para isolar o subconjunto de interações relevantes para simular o erro.

4 AUTOMAÇÃO DA EXECUÇÃO DE TESTES DE *SOFTWARE* EMBARCADO

A automação é uma solução para reduzir de envolvimento humano nas atividades manuais e repetitivas. No contexto de execução de testes, a automação ocorre em decorrêcia do volume de testes que devem ser realizados ao longo da produção de um sistema.

Segundo Pfleeger (PFLEEGER, 2004) *"[...] o teste sempre envolverá o esforço manual requerido para rastrear um problema até sua causa original. A automação auxilia, mas não necessariamente substitui a função humana.*

O processo de execução de testes consiste em executar a implementação em paralelo com os testes, sincronizando a ação de entrada da aplicação com a saída do teste (GAUDEL, 2010). O sucesso ou falha da execução do testes ocorre em função das observações feitas: ações realizadas, resultados obtidos, ausência de resultados, etc.

4.1 MODELO CONCEITUAL E ARQUITETURA

AUTETESE é um ambiente de execução automática de testes de *software* embarcado. A execução dos testes é realizada com o intuito de verificar se o *software* está em conformidade com sua especificação. Em função disto, assume-se que os dados utilizados como entrada para o ambiente estão corretos. A corretude do modelo de dados pode ser garantida através de mecanismos como, por exemplo, revisões ou inspeções (PRESSMAN, 2011).

O AUTETESE possui cinco fases conceituais: (i) importar informações do modelo e casos de teste; (ii) executar os testes de conformidade com a especificação através de testes caixa preta; (iii) executar os testes estruturais se foram encontrados erros na fase anterior; (iv) disponibilizar interação com o ambiente de depuração no caso de sucesso dos testes estruturais; (v) sintetizar os resultados dos testes em um relatório.

A primeira fase tem como objetivo extrair os dados necessários sobre os requisitos do sistema. Eles são utilizados para configurar a excução do *software*, a emulação do *hardware*, os casos de teste do sistema, entre outros. O usuário do ambiente é o responsável pelo preenchimento correto do arquivo de configuração de AUTETESE, que deve conter requisitos do sistema e a variedade de valores que este requisito pode assumir. Adicionalmente, são definidas neste arquivo as customizações das funcionalidades do ambiente, como: quantos testes serão executados, se o ambiente executará os testes es-

truturais, se haverá interação com o usuário para depuração, entre outros.

Vale ressaltar que a variação de requisito no escopo de AUTETESE é entendida como o intervalo de valores válidos para determinada configuração, por exemplo: a quantidade de número de núcleos de processamento, tamanho da memória, arquitetura alvo, tipo de escalonamento, ou qualquer particularidade do *software* embarcado.

Durante a fase de execução dos testes de conformidade, é da competência do ambiente proposto executar os testes caixa preta e relatar para quais combinações de requisitos obtiveram êxito e para quais houve falha. Isto significa que se novas funcionalidades forem introduzidas ou se o sistema for alterado, o ambiente acusará se algum requisito deixou de ser atendido. Não está na competência do ambiente a integridade dos testes mas, sim, relatar se o *software* mantém sua execução correta mesmo que haja variação dos requisitos, plataformas, arquiteturas, etc.

As fases 3 e 4 são opcionais e devem ser customizadas pelo usuário durante a primeira fase. Se o ambiente estiver configurado para executar os testes funcionais (fase 3), o usuário obrigatoriamente deve disponibilizar um arquivo contendo os comandos necessários para a execução destes testes. Uma ferramenta, cujo objetivo original é depuração, teve seu uso adaptado para executar os testes estruturais. Portanto, cabe ao usuário traduzir os passos do teste estrutural para a linguagem do depurador escolhido.

A fase 4 foca em disponibilizar interação com uma ferramenta de depuração, para que seja possível investigar o *software*. AUTETESE não contempla a automação da depuração do *software*, apenas sua execução automática. Cabe ao usuário informar no arquivo de configuração da fase 1 se deseja realizar a depuração de forma manual ou com execução automática dos comandos de depuração. Se configurado como intervenção manual, o ambiente disponibiliza acesso para a entrada de comandos de depuração por parte do usuário. Caso contrário, o usuário deve obrigatoriamente configurar o ambiente com um arquivo contendo os comandos necessários para a depuração.

Em ambos os casos o usuário é o responsável pela geração dos comandos de depuração do *software*, mas existem ferramentas que podem auxiliar na depuração, como a gerarão automática de *breakpoints* (ZHANG et al., 2013) e o controle do fluxo de execução (CHERN; VOLDER, 2007).

A última etapa da exeução do ambiente é responsável por sintetizar as informações obtidas durante a execução das fases anteriores. As informações são entregues em um relatório que contém: resumo quantitativo dos testes executados (total, sucesso e erro), identificação dos testes que evidenciaram o erro, *log* obtido através dos testes estruturais e *trace* da execução do *software*.

Em contraste com o modelo conceitual do ambiente, a arquitetura de AUTETESE é mais enxuta e contém apenas quatro fases. A Figura 3 ilustra

uma visão macro da arquitetura, dos passos realizados e das saídas de cada fase presente no processo de automação da execução de testes de *software* com AUTETESE.

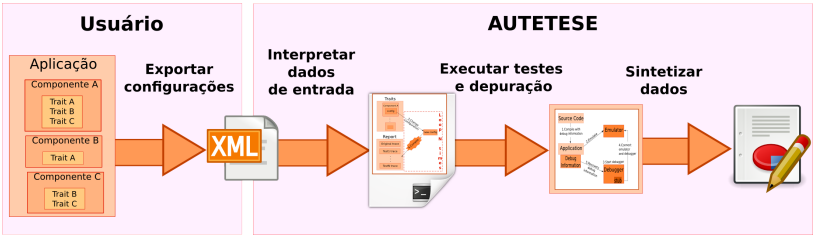

Figura 3 – Visão geral da arquitetura de AUTETESE

O processo é iniciado após o usuário disponibilizar um arquivo XML contendo dados relevantes para a execução dos testes da aplicação. Neste arquivo de configuração deve estar discriminado em qual *software* deseja-se realizar os testes, quais as funcionalidades a serem verificadas e os valores que podem assumir. A Seção 4.2 detalha melhor todas as questões de preenchimento correto do arquivo XML.

O próximo passo ocorre dentro de AUTETESE, no qual o ambiente importa os dados do XML de configuração e traduz os casos de teste em scripts de execução do *software*. A Seção 4.3 possui mais informações sobre a idealização e implementação do algoritmo de execução de testes.

Os testes funcionais, estruturais e a depuração do *software* são executados em um ambiente composto por ferramentas cuja finalidade inicial é depuração. Estas ferramentas foram adaptadas para auxiliar na execução de testes, como pode ser visto na Seção 4.4. A ordem de execução é: testes funcionais, testes estruturais e depuração. As duas últimas etapas são opcionais, de acordo com a configuração inicial.

Por fim, os resultados da execução dos casos de testes são sintetizados em um relatório apresentado ao usuário. As informações sobre a disposição dos dados e os tipos de resultados apresentados pelo relatório são descritas na Seção 4.5

4.2 EXPORTAR CONFIGURAÇÕES

A configuração de AUTETESE é feita através de um arquivo XML. O uso da linguagem XML se deu pela facilidade e legibilidade para definir todas as regras necessárias para executar os testes. O ajuste inicial é manual e simples,

uma vez que este arquivo pode ser lido quase como um texto.

A Figura 4 traz um exemplo do arquivo de configuração para uma aplicação chamada "*exemplo*". Dentro do arquivo estão descritas duas configurações: a primeira é identificada como "*ARCH*"e pode assumir os valores "*IA*32"ou "*AVR*8", já a segunda propriedade está relacionada com a depuração e é um arquivo que contém *breakpoints* e está no seguinte caminho: "*/home/breakpoints.txt*".

```
<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 4 – Exemplo do arquivo de configuração do teste para a TAP.

No arquivo de configuração de AUTETESE, cada execução de teste deve iniciar com a *tag* `<test>`, que identifica que um novo caso de teste será especificado. A partir desta *tag* raiz, é possível acrescentar as opções de ajustes na execução do teste. Elas são:

<**application**> - esta *tag*, juntamente com o descritor `name`, deve ser preenchida com o nome do *software* no qual serão executados os testes. No caso da aplicação da Figura 4, a *tag* é a `<application name="exemplo">`. É possível apontar mais de um SUT em um mesmo arquivo, mas cada um deve possuir seus próprios parâmetros de execução. Ou seja, todas as definições de `<application>` devem ser filhas de uma *tag* `<test>`.

<**configuration**> - é uma *tag* agrupadora para todas as configurações de um caso de teste. Com esta separação estrutural é possível especificar vários casos de teste para uma mesma aplicação.

<**trait**> - é a *tag* onde define-se o parâmetro do sistema que será analisado. Ela possui o descritor `id` para armazenar o nome da configuração e o descritor `scope` para os casos em que o parâmetro deve ser verificado dentro de um escopo específico da aplicação. Por exemplo, o valor de uma fatia de tempo (`QUANTUM`) pode ser aplicado em vários contextos: *threads*, *CPUs*, etc. Para considerar apenas o contexto de

threads, expressa-se da seguinte maneira: `<trait scope="Thread" id="QUANTUM">`

<min>, <max>, <value> - são *tags* filhas de `<trait>` e são utilizadas na definição dos valores dos parâmetros. As *tags* `<min>` e `<max>` representam, respectivamente, o valor mínimo e máximo que a cofiguração pode atingir, ou seja, qualquer valor neste intervalo é considerado válido. Já a *tag* `<value>` identifica um possível valor para a configuração. Caso o usuário deseje discriminar todos os possíves valores para uma configuração, cada um deles deve ser apresentado em uma nova definição de `<value>`. Pode-se usar como exemplo a definição de valores para a configuração `ARCH` da Figura 4, descritas como `<value>IA32</value>` e `<value>AVR8</value>`.

<structural>, <debug> - são as *tags* que definem onde encontram-se, respectivamente, o arquivo com os testes caixa branca e o arquivo de depuração. O caminho até o arquivo é representado em `<path>` e deve ser definido com o caminho absoluto do arquivo, ou seja, inicia-se com a referência do diretório raiz (/) e a partir dele apresenta uma árvore de diretórios até encontrar o arquivo. No caso da *tag* `<debug>`, o usuário pode optar pela forma manual, configurando a *tag* sem um arquivo de depuração. A Figura 4 apresenta um exemplo da *tag* `<debug>`, com referência para o arquivo `breakpoints.txt`.

AUTETESE oferece três granularidades de configuração para o teste caixa preta: aleatória, parcialmente aleatória e determinada. Elas são selecionadas de acordo com a quantidade de informações fornecidas pelo usuário sobre os casos de teste e das características do SUT. São elas:

Execução aleatória - utilizada para testar valores fora do convencional, a fim de verificar a robustez da aplicação. Nela o usuário informa somente os requisitos do sistema através dos arquivo de *traits*. Cabe então ao ambiente inferir os requisitos à partir do arquivo de *traits* e gerar valores válidos, respeitando sua tipagem. A geração destes valores não segue o modelo do *software*, apenas gera valores arbitrários para requisitos aleatórios.

Este teste foi desenvolvido como um pior caso para comparações estatísticas, pois o ambiente pode gerar testes repetidos e fora do limite de modelagem do *software* ocasionando distorções nos resultados obtidos.

A execução aleatória possui utilidade quando sabe-se que o *software* possui erros e não existem indícios para descobrir a origem deste erro

ao iniciar a depuração. Através deste tipo de execução pode-se encontrar valores errados de configuração e auxiliar os desenvolvedores com pouco conhecimento dos requisitos do *software* a depurar pequenas aplicações.

Execução parcialmente aleatória - utilizada para executar testes em que apenas algumas informações são fornecidas à priori e para verificar os requisitos do sistema que não possuem um valor determinado, ou seja, mais de um valor pode ser considerado correto. Neste caso, o usuário informa um intervalo de valores válidos para os casos de teste na configuração.
Por exemplo, sabe-se que o requisito *X* pode assumir valores de 1 a 10, sendo assim, o ambiente seria configurado da seguinte maneira:

```
<trait id="X">
        <min>1</min>
        <max>10</max>
</trait>
```

Se apenas o requisito for informado, cabe ao ambiente inferir os valores dos requisitos à partir do arquivo de *traits*, assim como na execução aleatória. Desta forma, o ambiente não garante que os valores gerados sejam distintos uns dos outros e existe a possibilidade da execução de testes apresentar resultados com falsos negativos.

Execução determinada - utilizada quando os casos de teste estão bem definidos e todos os requisitos podem ser traduzidos no arquivo de configuração. Se algum caso de teste na execução determinada obtiver valores diferente dos descritos, AUTETESE considerará esta execução falha.
Por exemplo, para que o requisito *X* assuma apenas os valores 1 ou 10 o ambiente deve ser configurado da seguinte maneira:

```
<trait id="X">
        <value>1</value>
        <value>10</value>
</trait>
```

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.

A diferença fundamental entre os três modos de execução dos testes funcionais é a quantidade de regras que são inseridas no arquivo de configuração:

o modo aleatório não possui regras, o parcialmente aleatório contém definição de valor ou propriedade a ser considerada, e o modo determinado possui tanto a propriedade quanto os valores a serem trocados.

AUTETESE também suporta a execução de testes estruturais e depuração de maneira primária. Os testes de caixa branca são importados através do arquivo de configuração, que contém uma sequência de comandos que representam os casos de teste. Para a depuração, o usuário pode optar pela forma manual ou execução automatizada. Na forma manual, AUTETESE fornece ao usuário a interação com o depurador, na qual será possível inserir os comandos desejados. Já na depuração automatizada, o sistema é configurado com um arquivo que contém uma sequência de passos de depuração. Um exemplo deste tipo de arquivo utilizando a ferramenta *GNU Project Debugger* (GDB) é ilustado na Figura 5.

```
1 set logging on
2 break main
3 continue
4 continue
```

Figura 5 – Exemplo do arquivo de breakpoint.

A sequência de passos inicia-se ao ativar o *log* do GDB (*set logging on*) para poder salvar todos os passos da depuração e poder analisá-los posteriormente, caso necessário. Com esta opção ativa no *script* de depuração os passos executados no GDB também serão anexados ao relatório final de AUTETESE. O comando *break* adiciona *breakpoints* nas funções desejadas, neste exemplo é adicionado na função *main*. O comando *continue* é um sinal de liberação da execução do sistema para iniciar a depuração.

4.3 INTERPRETAR DADOS DE ENTRADA

Na fase de interpretar os dados de entrada ocorre o processamento das informações e a transformação dos dados em ações a serem executadas de maneira automática. Para realizar este processo AUTETESE utiliza-se de um algoritmo que extrai as informações necessárias para a realização dos testes e realiza a troca de valores de configuração.

O algoritmo de troca de parâmetros, conforme ilustrado na Figura 6, verifica as configurações disponíveis no arquivo XML e seleciona as trocas a serem realizadas, de acordo com a especificação. O valor da propriedade é modificado diretamente no arquivo de configuração da aplicação (*traits*) e é compilado e executado em cada nova troca de parâmetros.

A troca de parâmetros é realizada a partir de uma combinação en-

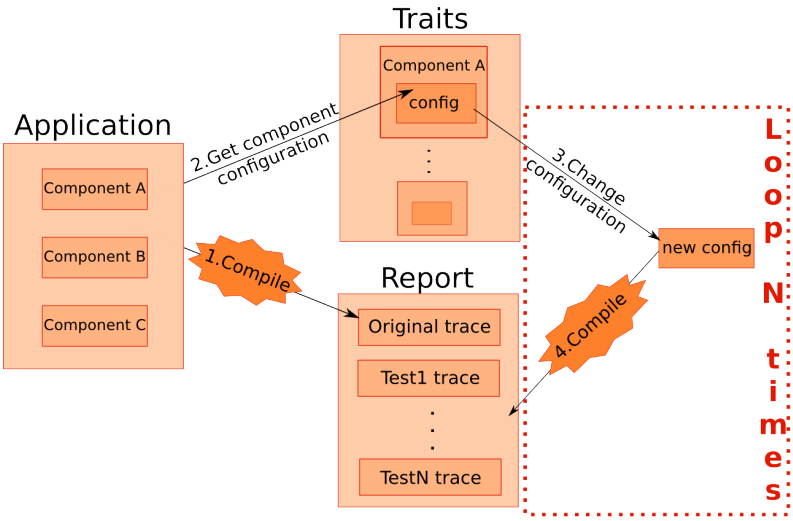

Figura 6 – Algoritmo para troca de valores de configuração do *software*

tre as propriedades e seus possíveis valores, de acordo com a fórmula de combinação matemática: $C_s^n = \binom{n}{s} = \frac{n!}{s!\cdot(n-s)!}$. Sendo n a quantidade de formas distintas em que é possível escolher os elementos de um conjunto de entrada, e n é o tamanho do conjunto de entradas.

No caso de AUTETESE, o tamanho do conjunto de entradas varia de acordo com o arquivo de configuração do ambiente. Por exemplo, quando a configuração aponta para uma execução determinada, o número de entradas é igual à quantidade de $tags < value >$ da configuração.

O tipo de execução de testes presente no XML possui grande influência na troca dos parâmetros. No caso de uma execução aleatória, em que apenas o arquivo *traits* é informado, o algoritmo deve selecionar tanto uma configuração quanto um valor para realizar a troca. Já em casos de execução determinada, o algoritmo limita-se a realizar as trocas propostas. Na execução parcialmente aleatória ambas as condições estão presentes, sendo assim, este tipo de execução será utilizada para exemplificar a troca de parâmetros.

Suponha a execução de testes para a aplicação exemplo que contém as propriedades: (i) `Escalonador`, que assume valores fixos; e (ii) `NumeroThreads`, cujo valor é variável. A Figura 7 apresenta um exemplo do arquivo de configuração XML e das aplicações originadas à partir da troca de parâmetros.

Para a aplicação `Exemplo`, a combinação realizada foi:
$C_1^2 \cdot C_1^4 = \frac{2!}{1!\cdot(2-1)!} \cdot \frac{4!}{1!\cdot(4-1)!} = 8$

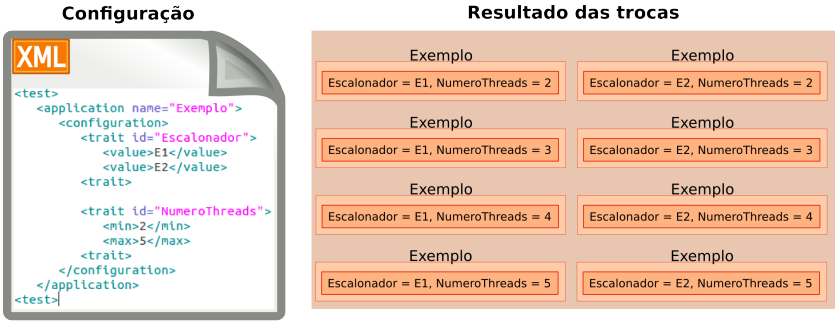

Figura 7 – Configuração de AUTETESE e os resultados das trocas de valores de configuração da aplicação `Exemplo`

O conjunto de resultados obtidos é referente à combinação dos dois possíveis valores para `Escalonador` com a combinação dos quatro valores para propriedade `NumeroThreads`, resultando em 8 variações de configuração, que serão compiladas e executadas posteriormente.

4.4 EXECUTAR TESTES E DEPURAÇÃO

AUTETESE possui suporte para executar tanto os testes funcionais, quanto os estruturais. Adicionalmente, se ambos os testes apresentarem falha na execução, inicia-se a execução da depuração. A Figura 8 ilustra o fluxo das atividades de teste.

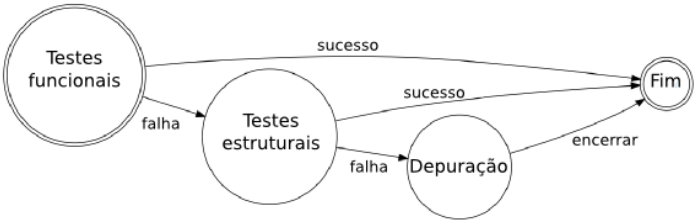

Figura 8 – Fluxo de execução dos testes e da depuração da aplicação com AUTETESE

A execução dos testes funcionais é realizada automaticamente após obter as aplicações compiladas pelo algoritmo de troca de parâmetros. Já a execução dos testes estruturais ocorre somente em caso de falha na execução

dos testes funcionais e se houver sido explicitada a configuração no arquivo XML do ambiente.

A depuração da aplicação ocorre após as execuções de ambos os teste e, somente se, resultarem em falha. Este processo é opcional e pode ser configurado no XML como manual ou automático. Na depuração manual, o ambiente fornece uma entrada de dados para que o usuário possa depurar a aplicação. Para a depuração automática, o arquivo XML deve conter o caminho para um outro arquivo que contém um roteiro da depuração e AUTETESE apenas executará os comandos que estiverem neste roteiro.

O projeto do ambiente para que testes e depuração compartilhem uma infraestrutura iniciou-se a partir da observação e estudo de vários ambientes de execução de ambas atividades em separado. O desenvolvimento de AUTETESE focou no suporte para sistemas embarcados, pois estes sistemas geralmente possuem apenas os recursos essenciais para desempenhar sua função e as atividades de teste e depuração compartilham os escassos recursos do SUT.

A execução dos testes e depuração em AUTETESE é realizada utilizando-se um emulador e um depurador. A emulação do sistema possibilita a visualização do estado do *software* e proporciona maior liberdade para controlar a execução da aplicação. Sua função principal para o ambiente é auxiliar o teste do *software* embarcado sem o *hardware* físico.

A função do depurador é de auxiliar em encontrar os erros do *software*, facilitando sua correção. Eventualmente, o ambiente de depuração também será utilizado como ferramenta auxiliar para os testes de caixa branca, visto que sua conexão com o emulador fornece dados atualizados sobre o sistema em execução.

4.5 SINTETIZAR OS DADOS

A síntese dos dados e emissão do relatório para o usuário ocorre após a execução dos casos de teste e da depuração do sistema. Este documento apresenta diferentes configurações que dependem do tipo de execução dos testes. A Figura 9 apresenta um exemplo dos tipos de relatórios que podem ser gerados por AUTETESE.

No caso de uma execução determinada, o usuário sabe quais configurações foram trocadas e por quais valores. Desta forma, o relatório não precisa explicitar todas as trocas realizadas, apresentando apenas um resumo da execução.

Para a execução parcialmente aleatória sabe-se a configuração que será trocada, entretanto, não conhece os valores, uma vez que estes foram gerados por AUTETESE. Sendo assim, no relatório são informados todos os valores

Execução determinada

```
.*.*.*.*.* TEST REPORT .*.*.*.*.*
= Configurations =
QUANTUM: 10000, 20000, 50000, 100000
THREADS: 1, 2, 4, 8, 16, 32

APPLICATION = exemplo1
Successful tests:16 - Failed tests:8
Failed tests execution.
Log and debugging information on attachment.
```

Execução parcialmente aleatória

```
.*.*.*.*.* TEST REPORT .*.*.*.*.*
APPLICATION = exemplo1

ORIGINAL TRAITS LINE = echo_reply = true;
VALUES= true, false

ORIGINAL TRAITS LINE = THREADS = 3;
VALUES= 6, 67, 72, 40, 10

Successful tests:4 - Failed tests:6
Failed tests execution.
Log and debugging information on attachment.
```

Execução totalmente aleatória

```
.*.*.*.*.* TEST REPORT .*.*.*.*.*
APPLICATION = exemplo1

ORIGINAL TRAITS LINE = echo_reply = true;
MODIFYED TRAITS LINE = echo_reply = false;

ORIGINAL TRAITS LINE = THREADS = 3;
MODIFYED TRAITS LINE = THREADS = 24;
------------------ FAILURE

ORIGINAL TRAITS LINE = QUANTUM = 10000;
MODIFYED TRAITS LINE = QUANTUM = 30109;
```

Figura 9 – Exemplo de apresentação do relatório conforme tipo de execução dos testes.

utilizados nas trocas.

No caso do relatório para uma execução aleatória, devem ser descritas todas as configurações que foram trocadas pelo ambiente, juntamente com seus respectivos valores e o resultado da execução.

Todos os relatórios possuem um resumo da quantidade de testes executados, quantos obtiveram sucesso e quantos falharam. Um arquivo de *log* de execução é armazenado para cada falha da aplicação na execução dos testes. Este *log* fica anexado ao relatório e contém os dados específicos sobre a compilação, execução dos testes (funcionais e estruturais) e depuração.

4.6 IMPLEMENTAÇÃO DE AUTETESE

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 as 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 e 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 são as adaptações e novas implementações de componentes de *software* ou *hardware* que atendem a 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, AUTETESE procura atender estes requisitos ao extrapolar os conceitos da metodologia de projeto orientado aplicação e da técnica 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 podemos 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 troca de parâmetros de configuração do sistema 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.

4.6.1 Troca de parâmetros de configuração do sistema

TAP 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 compartilham uma mesma especificação base, especializando apenas a configuração desejada. Desta forma, é possível executar o mesmo conjunto de testes para atender uma maior variabilidade de configurações de 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 e a partir das especificações contidas nele, o algoritmo flui no sentido de tentar encontrar a característica desejada, trocá-la por um 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 nas trocas realizadas. Se o aquivo de configuração não trouxer sugestões de propriedades a serem modificadas e valores cuja semântica é adequada, a ferramenta apenas selecionará valores estruturalmente válidos.

4.6.2 Detalhes da implementação

A implementação do algoritmo utiliza como base um sistema operacional com uma grande capacidade de configuração do sistema. O 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 (LISTA..., 2014).

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.

Cada aplicação gerada possui um arquivo próprio de configuração de

Algoritmo 1: Algoritmo de troca dos parâmetros de configuração

```
   Entrada: arquivo de configuração do ambiente
   Saída: relatório com resultado da execução dos testes funcionais
 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;
```

abstrações para definir o seu comportamento. A Figura 10 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 10 – Trecho do *trait* da aplicação DMEC.

Como a execução do algoritmo de troca de parâmetros ocorre dentro do ambiente de AUTETESE, os *traits* desta aplicação devem ser adaptados para o arquivo de cofiguração de XML do ambiente. 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.

4.6.3 Ambiente compartilhado de testes e depuração

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 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, testar o sistema com um protótipo limitado do *hardware*, ou até utilizando o próprio ambiente de desenvolvimento (JGRENNING, 2004).

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; MARGI et al., 2009; DANTAS et al., 2010).

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 uma emulação completa do sistema, que possibilite 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 heterogêneos 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.

4.6.4 Detalhes da implementação

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 *Quick Emulator* (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 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 11 apresenta as atividades necessárias para executar a depuração remota em conjunto com a simulação dos testes.

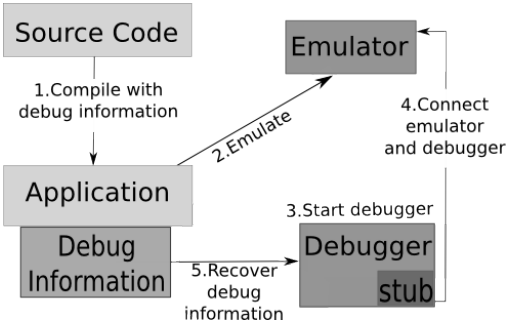

Figura 11 – Atividades de integração entre emulador e depurador

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

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.

4.6.5 Considerações 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*.

Estas alterações podem, em alguns casos, gerar erros que provavelmente não existiriam em uma compilação com as otimizações. Isto implica em falsos positivos que podem a vir ser apontados por TAP. Porém, os relatórios gerados por TAP, independende de falsos positivos, necessitam de interpretação. Isto é, a ferramenta não substitui o desenvolvedor, ela auxilia no desenvolvimento, e cabe ao desenvolvedor considerar ou não mudanças de acordo com o relatório. Em suma, é uma questão de hermenêutica.

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, existem trabalhos que discutem como as otimizações podem influenciar *softwares* com múltiplas *threads* (JIA; CHAN, 2013) e sugestões (EFFINGER-DEAN et al., 2012; FONSECA et al., 2010; KOUWE; GIUFFRIDA; TANENBAUM, 2014) para o teste e depuração de *software* com o mínimo de influência no sistema, com foco em não ocultar erros de simultaneidade.

5 RESULTADOS EXPERIMENTAIS E ANÁLISE

Para validar o ambiente proposto, foi desenvolvida uma ferramenta para a automação da execução da a troca de parâmetros de configuração, que utiliza o ambiente compartilhado para a execução dos testes funcionais, estruturais e da 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 códigos fonte gerados por desenvolvedores com pouca experiência em implementação e manutenção de *software* embarcado e seu foco é na viabilização da ferramenta como auxiliadora no aprendizado.

5.1 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 12 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.

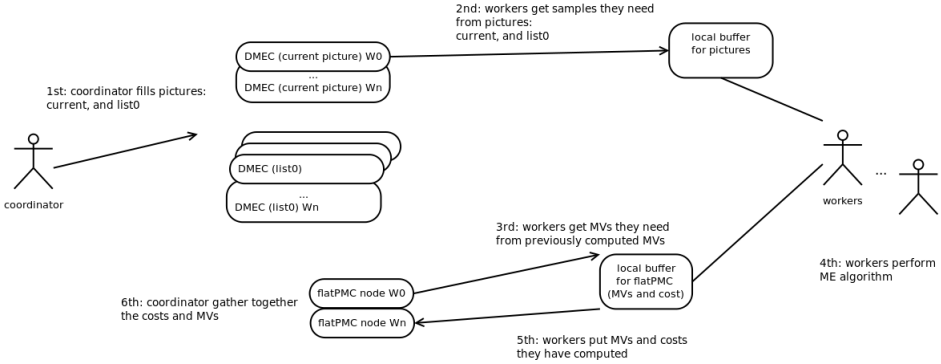

Figura 12 – Interação entre o coordenador e os trabalhadores na aplicação teste do DMEC (LUDWICH; FROHLICH, 2011).

5.1.1 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 13 e na Figura 14.

As Figuras 13 e 14 mostram que foi possível 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, mais adiante discute-se o porquê. O aprendizado que pode-se tirar destes casos é que apenas compilar o código não garante uma aplicação livre de *bugs*.

Para os casos nos quais a execução da aplicação não foi bem suce-

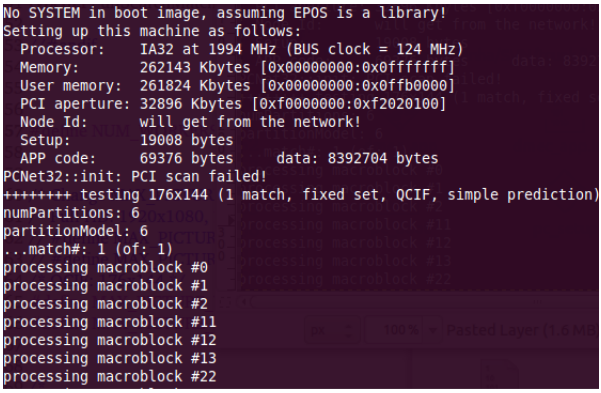

Figura 13 – Teste do DMEC com configuração `NUM_WORKERS` = 6

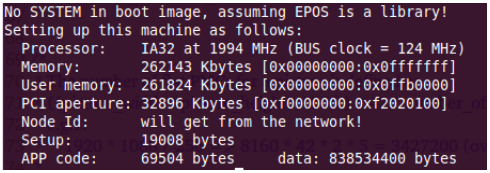

Figura 14 – Teste do DMEC com configuração `NUM_WORKERS` = 60

dida, TAP automaticamente aciona o depurador para que se possa descobrir 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 15.

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. Isto 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.

Após a execução da ferramenta e análise do relatório, foi possível determinar que sempre que a configuração `NUM_WORKERS` apresentava um número maior que 6 a aplicação se comportava de maneira anômala, surgindo a hipótese de que existia um limite máximo de trabalhadores.

Descobriu-se que, embora o particionamento da imagem seja fixo, dependendo do número de *threads* era realizado um particionamento das ima-

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

gens diferente. Para um número de *threads* maior que 6 havia uma falha na aplicação devido a um erro cometido pela *thread* coordenadora e que impossibilitava que o SUT atingisse a função `main`.

A partir desta descoberta foi possível maximizar o paralelismo das estimativas sem que houvesse efeitos colaterais no funcionamento da aplicação. Esta informação ajudou no desenvolvimento do componente, uma vez que descobriu-se que 6 *threads* é o limite máximo de *threads* suportado por este tipo de particionamento, e que para supotar um número maior que este seria necessário criar novas formas de particionamento.

Apesar da ferramenta ter fornecido dados importantes para verificar uma falha na execução do SUT, vale lembrar que a qualidade da informação de retorno é inerente à qualidade de informação à priori que é repassada na configuração de TAP. A Figura 16 apresenta um trecho de relatório com algumas configurações geradas.

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 17 e 18 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

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

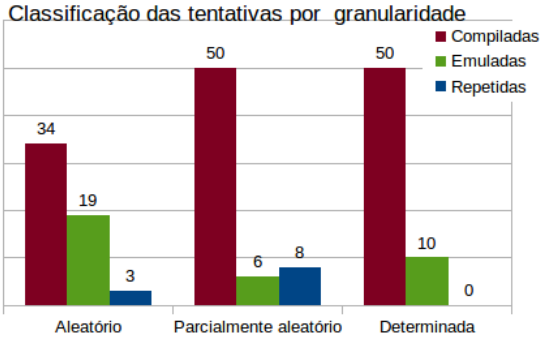

Figura 17 – Classificação das tentativas realizadas versus a configuração da granularidade.

alterada esta mesma propriedade com valores de 1 a 60.

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 19, 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

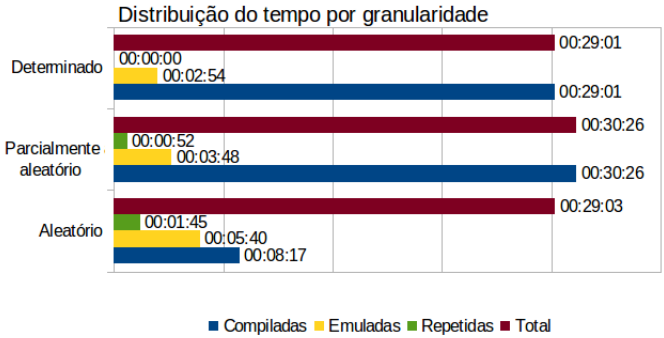

Figura 18 – Classificação das tentativas realizadas versus o consumo de tempo.

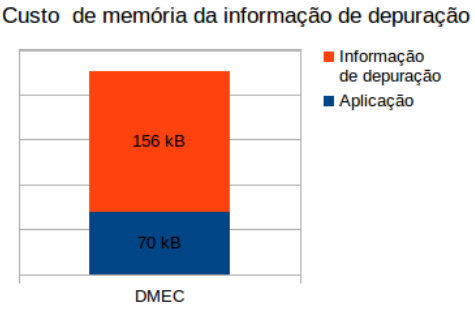

Figura 19 – Consumo de memória extra para armazenar as informações de depuração.

embarcado real, o tamanho desta nova imagem seria proibitivo.

5.2 EXPERIMENTO 2 - UTILIZAÇÃO EM DISCIPLINA E EFETIVIDADE DOS RESULTADOS

Este experimento foi elaborado para verificar a utilização da ferramenta como artifício para estudo de disciplinas que envolvem codificação de *software* embarcado. Além disso, este experimento tem como objetivo secundário a avaliação da ferramenta e da efetividade dos seus resultados.

Para verificar a viabilidade do uso em disciplinas o experimento foi aplicado em códigos fonte de desenvolvedores inexperientes na produção de

software embarcado. A ideia é entender como a ferramenta pode ser introduzida em disciplinas de desenvolvimento de *software* considerando a facilidade de uso, abrangência dos testes e ajuste fino do relatório gerado.

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, desenvolvidos no 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.

5.2.1 Configuração do experimento e execução dos testes

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*. As propriedades do arquivo de configuração desenvolvido para a disciplina podem ser observadas na Tabela 1.

A configuração da ferramenta para a disciplina focou em algumas configurações, mas com alta variabilidade de valores candidatos para a substituição. TAP foi configurada desta forma para que seja possível apresentar aos alunos a quantidade de aplicações que devem funcionar corretamente após a implementação de um determinado exercício.

Quando o foco é o aprendizado, é justo que esta configuração seja bem abrangente, visto que esta variedade pode ampliar a visão para cenários que talvez sejam importantes e que podem ter sido negligenciados durante a fase de concepção, incluindo a idealização dos requisitos não funcionais.

Inicialmente o EPOS conta com 18 aplicações de teste e em cada uma

Tabela 1 – Configuração de TAP para as aplicações do EPOS da disciplina.

	Namespace		
Propriedades	**Aplicação**	**Sistema**	**Thread**
STACK_SIZE	512MB	512MB	-
	1GB	1GB	-
HEAP_SIZE	512MB	512MB	-
	1GB	1GB	-
QUANTUM	-	10ms	10ms
	-	20ms	20ms
	-	50ms	50ms
	-	100ms	100ms
	-	200ms	200ms
	-	500ms	500ms
Scheduling_Criteria	-	-	RM
	-	-	EDF

foram realizadas 648 trocas, ou seja, TAP trocou as configurações destas aplicações e gerou um total de 11.664 variações, as quais foram automaticamente executadas e depuradas. Para cada aplicação (e suas variações) foram necessárias cerca de 20 horas de processamento.

Esta abrangência no resultado gera uma grande quantidade de dados, portanto, para simplificar o estudo da execução dos testes, o relatório fornecido por TAP apresenta informações resumidas, com a opção de verificar os arquivos de *log* das execuções falhas. Na Figura 20 encontra-se um trecho de relatório gerado pela ferramenta.

```
.*.*.*.*.* TEST REPORT .*.*.*.*.*
= Configurations =
Quantum: 10ms, 20ms, 50ms, 100ms, 200ms, 500ms
Stack: 512MB, 1GB
Heap: 512MB, 1GB
Schedulers: EDM, RM

APPLICATION = active_test
Successful tests:248 - Failed tests:400
Failed tests execution.
Log and debugging information on attachment.
```

Figura 20 – Trecho de relatório gerado por TAP para as aplicações do EPOS da disciplina.

Os arquivos com as execuções falhas contém informações desde o mo-

mento de compilação até o relatório final da depuração com o GDB. Cada execução tem seu próprio arquivo, identificado pelo próprio nome concatenado com as configurações que foram trocadas.

Utilizando como exemplo a Figura 20, nota-se que a aplicação *active_test* continha algumas execuções falhas, sendo assim, dentre os anexos devemos encontrar 400 arquivos cujo nome iniciam-se com esta aplicação. Por exemplo, o arquivo *active_test_Scheduler_EDF.log* é um relatório de falha da execução dos testes da aplicação *active_test* para quando a configuração *Scheduler* estava com o valor EDF.

Através dos dados é possível notar que o número de execuções com falha começa a diminuir conforme são entregues os exercícios.Este resultado era esperado se for considerado o aumento da familiaridade com o sistema operacional. Outro fato crucial para este resultado está na realização incremental dos trabalhos, em que o aluno precisa ter a implementação do trabalho n para entregar corretamente o trabalho $n+1$.

Houve uma a grande tendência de redução da quantidade de execuções com falha no início da disciplina, que começou a estabilizar e seguir com mais uma leve redução. Isto ocorreu porque os exercícios de $E1$ até $E3$ relacionam-se diretamente com o assunto *threads*, criando uma certa zona de estabilidade. Esta estabilidade se quebrou à partir do $E4$, cuja avaliação envolve o conceitos como o de eventos programados e rotina de tratamento de interrupções.

A distribuição dos erros através dos exercícios pode ser observada na Figura 21. Considerando os percentuais de falha fica claro que quando iniciou-se um assunto mais aprofundado houveram maiores dificuldades de aprendizado. Se focarmos em uma análise exercício por exercício, os relatórios de teste gerados por TAP podem servir para analisar tendências do aprendizado e ajudar o responsável pelas turmas a conduzir aulas focadas nas dificuldades dos alunos.

A partir dos erros mais comuns dentre os alunos da turma é possível gerar discussões saudáveis sobre a área de sistemas operacionais de modo a facilitar o aprendizado e desenvolvimento dos próximos exercícios práticos.

Como os experimentos possuem um escopo fechado, o relatório da execução dos testes pode ser utilizado para verificar se os requisitos dos exercícios foram atendidos. Entretanto, deve ser considerado que a ferramenta serve para auxiliar a identificar padrões dos erros e dificuldades na hora de implementar um *software* embarcado, mas é imprescindível a intervenção manual e o discernimento pessoal para utilizá-la corretamente.

Como planos futuros consideramos utilizar esta ferramenta de maneira integrada ao serviço de controle de versão (*svn*, *git*, *mercurial*, etc). Através disso os alunos podem receber relatórios que vão auxiliá-los durante o aprendizado e que servem também para a geração de *feedback* para a ferramenta.

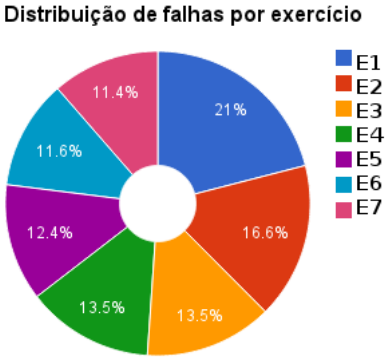

Figura 21 – Distribuição percentual das falhas.

Inclusive, este tipo de insumo é extremamente valioso para a melhoria da ferramenta em si pois os resultados gerados podem servir de base para realizar quantos ajustes forem necessários para manipular a execução dos testes e a depuração.

6 ANÁLISE QUALITATIVA DAS FERRAMENTAS DE TESTE E DEPURAÇÃO DE *SOFTWARE*

No Capítulo 3 foram discutidos 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.

Existem vários trabalhos (BERTOLINO, 2007; DURELLI et al., 2011; WIKLUND et al., 2012; RAFI et al., 2012) que apontam o passado, estado atual e futuro do teste e depuração de *software*. Pesquisadores como Bertolino (BERTOLINO, 2007) apontaram metas e desafios relevantes para o avanço da área, enquanto trabalhos mais recentes procuram definir o estado da arte, verificando quais desafios estão sendo atendidos e apontando as dívidas técnicas remanescentes.

Assim como o presente trabalho, grande parte da produção de conhecimento e pesquisa na área (mais de 46%) está focada em desenvolver ferramentas que automatizam uma ou várias atividades de teste de *software* (DURELLI et al., 2011).

Para que os trabalhos possam ser efetivamente comparados, a definição das características que serão analisadas qualitativamente foram inspiradas na pesquisa de 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.

6.1 METAS

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

6.1.1 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.

6.1.1.1 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.

6.1.1.2 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.

6.1.1.3 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.

6.1.1.4 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, con-

texto 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.

6.1.2 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.

6.1.2.1 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.

6.1.2.2 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.

6.1.2.3 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.

6.1.3 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*.

6.1.3.1 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.

6.1.3.2 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.

6.1.3.3 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.

6.2 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.

6.2.1 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 resul-

tados encontram-se na Tabela 2.

Tabela 2 – 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.

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 5 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 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 e 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 3 apresenta um resumo da análise comparativa entre TAP e as ferramentas correlatas no quesito teoria de teste universal.

Tabela 3 – 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.

6.2.2 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 4.

Tabela 4 – 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 5 – 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: Cap-

tura 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 6 apresenta um resumo da análise comparativa entre as ferramentas e técnicas correlatas para os desafios propostos pela meta de modelagem baseada em teste.

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.

Tabela 6 – 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

6.2.3 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 7 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.

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 e 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

Tabela 7 – 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.

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

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*.

Tabela 8 – 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

7 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 testes 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.

O segundo experimento comprovou a efetividade da utilização de TAP como uma ferramenta de auxílio em disciplinas de desenvolvimento de *software* embarcado. Com alguns ajustes é possível extrair relatórios que podem facilitar tanto os professores em suas avaliações, quanto os alunos em discussões e no seu aprendizado. Entretanto, em trabalhos futuros, se faz necessária a integração da ferramenta com serviçoes de versionamento e o envio automático dos relatórios para os alunos.

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.

7.1 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*.
- Criar interface de integração de TAP com serviços de versionamento de *software*.
- Criar mecanismo de envio de e-mail automático dos relatórios de TAP
- Propor a inclusão da ferramenta em disciplinas de desenvolvimento de *software* embarcado.

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 que corroborem a efetividade da ferramenta e que abranjam as novas funcionalidades.

REFERÊNCIAS BIBLIOGRÁFICAS

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

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.

BACH, J. Heuristics of software testability. 2003.

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>.

CHOWDHARY, V. Practicing testability in the real world. In: *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*. [S.l.: s.n.], 2009. p. 260–268.

DANTAS, A. V. M. et al. *Desro: um protocolo de roteamento com gerenciamento dinâmico de energia para redes de sensores sem fio*. Tese (Doutorado) — Dissertação-Universidade Federal de Viçosa-UFV, 2010.

DURELLI, V. et al. What a long, strange trip it's been: Past, present, and future perspectives on software testing research. In: *Software Engineering (SBES), 2011 25th Brazilian Symposium on*. [S.l.: s.n.], 2011. p. 30–39.

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.

FARRELL-VINAY, P. *Manage software testing*. [S.l.]: CRC Press, 2008.

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.

FRANZEN, M. B.; BELLINI, C. G. P. Arte ou prática em teste de software? *Revista Eletrônica de Administração*, v. 11, n. 3, 2005.

FREEDMAN, R. S. Testability of software components. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 17, n. 6, p. 553–564, jun. 1991. ISSN 0098-5589. <http://dx.doi.org/10.1109/32.87281>.

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

GAUDEL, M.-C. Software testing based on formal specification. In: *Testing Techniques in Software Engineering*. [S.l.]: Springer, 2010. p. 215–242.

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>.

HETZEL, B. *The Complete Guide to Software Testing*. 2nd. ed. Wellesley, MA, USA: QED Information Sciences, Inc., 1988. ISBN 0-89435-242-3, 9780471565673.

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.

IEEE Standard Classification for Software Anomalies. [S.l.], Jan 2010. 1-23 p.

IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, p. 76, Dec 1990.

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>.

KOSCIANSKI, A.; SOARES, M. d. S. *Qualidade de software*. [S.l.]: São Paulo: Novatec Editora, 2007.

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.

LISTA de publicações relacionadas ao EPOS. nov 2014. <http://www.lisha.ufsc.br/pub/index.php?key=EPOS>.

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.

MARGI, C. et al. Segurança em redes de sensores sem fio. *Livro de Minicursos do IX Simpósio Brasileiro em Segurança da Informaç ao e de Sistemas Computacionais (SBSeg 09)*, p. 149–194, 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.

NETO, A. C. D. Introdução a teste de software. *Revista Engenharia de Software Edição Especial, Artigo*, 2012.

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.

PEZZÈ, M.; YOUNG, M. *Teste e análise de software: processos, princípios e técnicas*. [S.l.]: Bookman, 2008.

PFLEEGER, S. L. Engenharia de software: teoria e prática. *2ª Edição, Prentice Hall*, 2004.

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.

RAFI, D. et al. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: *Automation of Software Test (AST), 2012 7th International Workshop on*. [S.l.: s.n.], 2012. p. 36–42.

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>.

WIKLUND, K. et al. Technical debt in test automation. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. [S.l.: s.n.], 2012. p. 887–892.

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.