UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Rita de Cássia Cazu Soldi

AUTOMAÇÃO DA EXECUÇÃO DE TESTES PARA SOFTWARE EMBARCADO

Florianópolis

2014

SUMÁRIO

1 INTRODUÇÃO

Sistemas embarcados podem ser apresentados como uma combinação entre *hardware* e *software* concebida para realizar uma tarefa específica. Estes sistemas estão amplamente acoplados a inúmeros dispositivos 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, a ruptura do oleoduto ou nos casos de overdose de radiação no tratamento de câncer (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 tragédias possui um relatório descrevendo como ocorreu o acidente e neles é possível verificar que a causa mais comuns é subestimar a complexidade do sistema e superestimar a eficácia dos testes (FROLA; MILLER, 1984). Mesmo depois de várias perdas relacionadas à erros em sistemas computacionais, ainda há muita complacência quando um sistema desvia do comportamento esperado, o que acaba subestimando a consequência deste erro. Para garantir que os componentes vão agir de acordo com o requisitado, uma boa prática é testar de maneira pragmática cada detalhe do sistema e sem subestimar a complexidade do mesmo.

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 destas variações. 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 deu espaço à novas possibilidades, 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). Então apesar de sua crescente complexidade, ainda são poucas as ferramentas de auxílio ao desenvolvimento de sistemas embarcados que focam em validação e verificação do *software*.

Ainda, constatou-se que as ferramentas disponíveis para automação da execução de testes não são de fácil compreensão/execução 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, fornecer um relatório com os testes já realizados e oferecer dicas para que se possa detectar o *bug*.

Trabalhos recentes apontam que mais de 80% dos erros de um sistema embarcado provém do *software*, não do *hardware*, e que tanto o teste quanto a depuração são de suma importância para a qualidade do projeto embarcado (TORRI et al., 2010). Desta forma, a motivação deste trabalho está em diminuir a dificuldade para realizar-se a verificação de 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 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 sólida da área de teste 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 das técnicas apresentadas neste trabalho.
- **Capítulo 3**: possui os trabalhos que representam o estado da arte da literatura e que servem de insperação para o desenvolvimento desta dissertação. Todos os trabalhos aqui apresentado se relacionam diretamente com o tema abordado.

2 CONCEITOS BÁSICOS

2.1 TESTE DE *SOFTWARE*

A área de pesquisa em teste de *software* é relativamente recente e a primeira conferência em que o tema foi discutido foi organizada em junho de 1972 (GELPERIN; HETZEL, 1988). Desde então há um esforço para definir melhor os conceitos relacionados à área e também para chegar-se a um consenso sobre a pontos chave como nomenclatura, documentação, especificação e execução de testes.

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 e quem todas elas estão funcionando corretamente.

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

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

- 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 tempo e esforço do retrabalho para adequar o *software*.

Os testes devem ser adotados desde o início do projeto de *software* a partir da definição dos requisitos a serem testados, escolha da abordagem utilizada, da política de testes, critérios para a conclusão do teste, entre outros. Para que o processo de teste alcance seu objetivo é necessária uma estratégia de testes, suficientemente flexível para modelar-se às peculiaridades do *software* e rígida o bastante para um acompanhamento satisfatório do projeto.

2.1.1 Classificação dos níveis de teste

As atividades de teste de *software* são normalmente classificados em níveis. Dentre os mais tradicionais estão a categorização baseada no processo de processo de *software* e a categorização baseada na maturidade de pensamento testadores(AMMANN; OFFUTT, 2008).

2.1.1.1 Modelo V

A classificação baseada no processo genérico do desenvolvimento do *software* e incentiva o projeto dos testes simultâneo com cada etapa de desenvolvimento, mesmo que o software só possa ser executado após a fase de implementação (ROOK, 1986). Nela cada atividade do processo de construção de um *software* é acompanhado de uma atividade de teste, ou seja, a informação de entrada do teste é normalmente derivada de uma atividade de desenvolvimento do *software*.

A Figura 1 apresenta um mapeamento típico entre as atividades *Modelo V*, nela é possível visualizar como cada tipo de teste se relaciona com o

processo genérico do desenvolvimento de *software*.

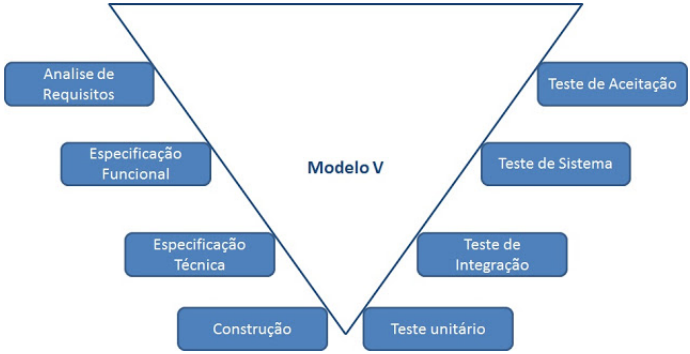

Figura 1 – ¿¿Vou colocar outra imagem, mas a ideia eh a mesma¿¿ Mapeamento entre as atividades de desenvolvimento e teste de *software*.

Análise de requisitos e Teste de aceitação: No desenvolvimento de *software*, a fase de análise de requisitos é quando se captura as necessidades do cliente, sendo assim, o teste de aceitação foi concebido para determinar se o *software* atingiu o objetivo e realmente satisfaz estas necessidades. Este tipo de teste geralmente são executados por um grupo restrito de usuários que simulam operações de rotina do sistema, de modo a verificar se seu comportamento está de acordo com o solicitado.

Especificação funcional e Teste de sistema - A fase de especificação funcional é quando arquitetura e os componentes são selecionados e, juntos, vão compor o *software*. Então, o teste de sistema é foi projetado para determinar se o sistema funciona conforme estipulado. Este nível de teste procura falhas no projeto e na especificação do *software* através de sua execução nos mesmos ambientes e nas mesmas condições que um usuário final o utilizaria.

Especificação técnica e Teste de integração - A fase de especificação técnica é voltada para o projeto da arquitetura do *software*, definindo a estrutura, comportamento e papel dos subsistemas que compõem a arquitetura geral do *software*. Seguindo a mesma linha, o teste de integração é projetado para avaliar se as interfaces entre subsistemas são consistentes e se a comunicação ocorre corretamente. Este teste geralmente é de responsabilidade dos membros da equipe de desenvolvimento e visa provocar falhas de comunicação entre os módulos integrados para construir o *software*.

Implementação e Teste de unidade - Implementação é a fase do desenvolvimento de *software* que realmente produz o código fonte. Portanto, o teste de unidade tem como alvo o código desenvolvido, ou seja, métodos, procedimentos (*procedures*) ou até mesmo mesmo pequenos trechos de código.

Tanto a implementação quanto o teste são de responsabilidade dos membros da equipe de desenvolvimento.

2.1.1.2 Modelo de maturidade

A classificação de níveis baseada na maturidade do processo de teste propõe que o nível dos testes seja derivado do objetivo do teste, que podem ser (BEIZER, 1990):

- **Nível 0** - Não há nenhuma diferença entre teste e depuração. Um teste com o nível zero não tem um propósito específico.

- **Nível 1** - possui o objetivo de mostrar que o *software* funciona. Normalmente este nível é utilizado quando a corretude do *software* ainda não foi alcançada/demonstrada, então é selecionado um conjunto de dados de teste que faz o *software* passar em determinados testes e falhar em outros. Apesar de ser uma técnica menos ingênua que o *Nível 0*, este tipo de teste pode não propiciar a descoberta de defeitos.

- **Nível 2** - possui o objetivo de mostrar que o *software* não funciona. Normalmente se seleciona um conjunto de dados para chegar pontos específicos do sistema como, por exemplo, checar a fronteira superior e inferior de um determinado dado de entrada. A transição do *Nível 1* para o *Nível 2* demanda cautela, pois apesar da descoberta de erros fazer parte linha raciocínio dos testadores, grande parte dos desenvolvedores possui linha de raciocínio mais próxima do nível anterior. Ou seja, esta mudança de nível pode gerar conflitos e afetar negativamente a moral da equipe.

- **Nível 3** - não possui o objetivo de reduzir o risco de utilizar *software*, sem provar que o *software* funciona ou não. Ao atingir a consciência de que testes não mostram ausência de falhas no *software* percebe-se também que sempre há um risco do *software* não funcionar corretamente. A partir deste nível, desenvolvedores e testadores compartilham o objetivo de reduzir o risco de utilizar o sistema, pois compreendem que embora o ideal seja testar todas as possibilidades, isto pode não ser possível.

- **Nível 4** - o teste é considerado como uma ferramenta que auxilia no desenvolvimento de um *software* com maior qualidade. O ato de testar não é um simples ato, mas sim uma maneira de produzir um *software* de baixo risco com pouco esforço. Neste nível, a geração de código é

realizada de maneira a suportar a concepção e o desenvolvimento de testes, facilitando também o reuso e manutenção do sistema.

2.2 DEPURAÇÃO DE *SOFTWARE* EMBARCADO

A depuração é a arte de diagnosticar erros em sistemas e determinar a melhor forma de corrigi-los. 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, que encontrou erros).

Erros podem ser encontrados por qualquer pessoa 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 *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. Para um suporte de depuração eficaz, existem alguns requisitos a serem seguidos (HOPKINS; MCDONALD-MAIER, 2006):

- O suporte à depuração não deve alterar significativamente o comportamento do sistema sob teste(*Software Under Test* (SUT)).

- Deve existir uma infra-estrutura de observação do estado interno do sistema e de possíveis pontos críticos.

- Garantia de acesso externo para controlar o estado interno do sistema e de seus recursos (incluindo periféricos).

- O impacto do custo do sistema deve ser o mínimo possível, 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 é independente de plataforma e pode ser 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.

2.2.1 Imprimir dados em dispositivos de saída

Um exemplo de instrumentação *software* é utilizar instruções de impressão para exibir/registrar variáveis, informações sobre o estado do sistema ou até o seu progresso através do código do programa. Esta técnica de depuração é um tanto primitiva e bastante intrusiva, pois normalmente está associada à mudanças no código fonte e recompilação do *software*.

Para realizar este tipo de depuração é necessário que o erro possa ser reproduzido e que ele não se altere com as impressões. Dependendo de como utilizada, esta técnica pode retardar a execução do sistema, modificando inclusive requisitos do próprio SUT.

Como a própria tentativa encontrar erros pode apresentar efeitos colaterais que mascaram o verdadeiro problema, torna-se difícil identificar erros relacionados com operações temporais, paralelismo e alocação de recursos.

2.2.2 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 fornece 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 dos 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).

2.2.3 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.

2.3 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 uma 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.

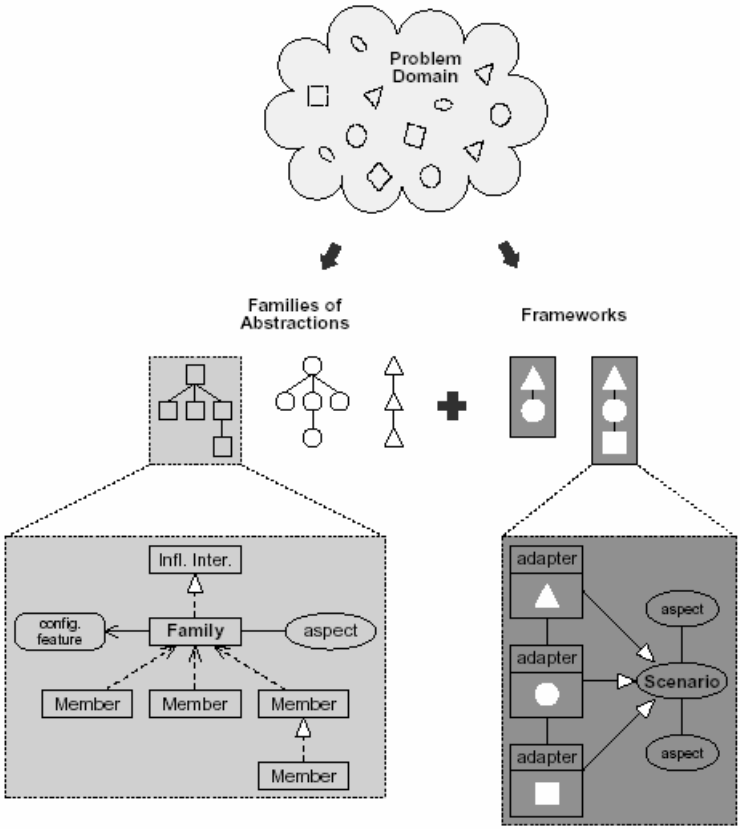

Figura 2 – Visão geral da metodologia de projeto de sistemas orientado à aplicação (FRÖHLICH, 2001)

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 de 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

Ferramenta de testes de *software* embarcado capaz de detectar automaticamente falhas 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*.
- **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 a tabela de símbolos para definir o sucesso ou falha do teste.

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.

Resumo

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.

Resumo

Ao explorar sistematicamente a automação de testes de *software* embarcado e componentizar cada operação necessária, *ATEMES* apresenta uma solução de grande desempenho para sistemas mutiprocessados. 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 4, 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, 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.

Resumo

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 5 será possível verificar uma grande semelhança no pro-

cesso 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 à partes do todo. Desta forma desenvolve-se a hipótese de integrar particionadoress de código-fonte nos ambientes de deupuração (**??**).

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, a estática ou a dinâmica.

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.

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

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

3.4.1 Partição do *software* com tamanhos de corte estático

(BINKLEY; GOLD; HARMAN, 2007)

3.4.2 Thin slicing

(SRIDHARAN; FINK; BODIK, 2007)

3.4.3 Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis

(CHEBARO et al., 2012)

Resumo

Vários conceitos ligeiramente diferentes das fatias de programas já foram propostos, bem como um número de métodos para calcular fatias. A principal razão para essa diversidade é o fato de que diferentes

Elas não constituem necessariamente um programa executável.

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 predicados: 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 falah 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 simultâneamente.

A implementação da extração de dados bidirecional executa um processo de votação coletiva iterativo, no qual todos os predicados tem um número que define a qualidade de representação. 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 hover convergência e quanto maior o número de predicados competindo pelo votos, menor é a quantidade de votos que cada um deles pode ter.

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

Resumo

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 estátistica baseada em amostras contribui com a discussão sobre a necessidade de ativar todos os possíveis estados do *sotware* 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 5 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 (**??**). A idéia do DD é chegar a um mínimo local, onde todos os eventos presentes interferem diretamete no comportamente 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 techo 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 excução 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 generérica 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* (**??**) 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 melorar 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 sub-árvores, 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 versõs 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* infinito, 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.

Resumo

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 interessane 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 5 será apresentado um algoritmo quem também se utiliza do delta para determinar sucesso/falha dos testes do SUT.

3.7 CAPTURA E REEXECUÇÃO

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

3.7.1 Selective capture and replay of program executions

(ORSO; KENNEDY, 2005)

3.7.2 Replaying and isolating failing multi-object interactions

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

3.7.3 Pinpointing interrupts in embedded real-time systems using context checksums

Um grande desafio nesta técnica é como se adaptar a interrupções, pois toda a estrutura de reprodução do *software* se baseia no fluxo de controle e uma interrupção tem a capacidade de romper esta sequência.

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.

3.7.4 Locating failure-inducing environment changes

(QI et al., 2011)

Resumo

4 O AMBIENTE COMPARTILHADO DE TESTE E DEPURAÇÃO

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

O ambiente utilizado desenvolvido para dar suporte ao presente trabalho conta com um emulador para simular a execução da aplicação e da automação dos testes. No caso de sucesso em um teste, ou seja, quando um erro é detectado, automaticamente uma ferramenta de depuração é iniciada e o resultado de todas as operações é repassado ao usuário.

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

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

4.1 CONFIGURAÇÃO DO AMBIENTE REMOTO

A integração da execução dos testes e da depuração é particular para cada máquina hospedeira e alvo, portanto, talvez alguns passos aqui apresentados devam ser adaptados dependendo de sua arquitetura alvo. A Figura 3 apresenta as atividades necessárias para executar a depuração remota em conjunto com a simulação dos testes.

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

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

Emular o sistema utilizando um emulador é um passo necessário para executar a aplicação na arquitetura alvo correta. Para executar este passo

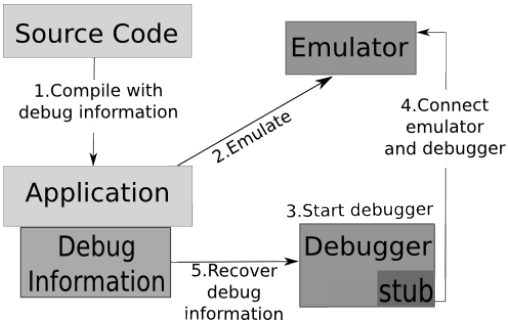

Figura 3 – Atividades de integração entre emulador e depurador

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: $qemu -fda\ app.img -serial\ stdio -s -S$

Conectar-se com o depurador começa com uma sessão do depurador inicializada em uma janela separada. 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: $target\ remote\ [endereço\_alvo] : [porta\_alvo]$

Recuperar as informações de depuração é um passo importante para ajudar os desenvolvedores a encontrarem erros. O arquivo gerado no primeiro item contém todas as informações que podem ser retiradas da compilação, como por exemplo o endereço de uma variável, ou os nomes contidos na tabela de símbolos. O arquivo usado para manter as informações de depuração deverá ser informado para o *GDB* usando o comando: $file\ [caminho\_do\_arquivo]$

Encontrar origem dos erros é uma atividade que depende do programa a ser depurado. A partir desta etapa, o desenvolvedor pode definir *breakpoints*, *watchpoints*, controlar a execução do programa e até mesmo permitir *logs*. Existem vários trabalhos com o foco em ajudar a encontrar os erros através da automação de alguns pontos, como a gerarão

automática de *breakpoints* (ZHANG et al., 2013) e o controle do fluxo de execução (CHERN; VOLDER, 2007).

5 A ARQUITETURA DE AUTOMAÇÃO DA EXECUÇÃO DE TESTES

A ideia de desenvolver a troca de parâmetros surgiu extrapolando-se conceitos da metodologia de projeto orientado aplicação (ADESD) e do uso de programação genérica para a área de testes. O projeto orientado à aplicação fornece um sistema embarcado desenvolvido a partir de componentes especificamente adaptados e configurados de acordo com os requisitos da aplicação alvo.

O fato de existir uma aplicação que fornece a certeza de que tudo que a compõe é essencial para seu funcionamento pode tornar o teste dos requisitos mais assertivo. Ainda, a programação genérica fornece uma maior adaptação do sistema às várias implementações de uma especificação.

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

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

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

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

Algoritmo 1: Algoritmo de Troca dos Parâmetros de Configuração

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

5.1 ABSTRAÇÕES NO *EPOS*

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

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

Cada aplicação tem seu próprio arquivo de configuração de abstrações para definir o seu comportamento. A Figura 4 mostra um trecho desta configuração para uma aplicação que simula um componente de estimativa de movimento para codificação $H.264$, o *DMEC*. Este trecho mostra como construir a aplicação, que neste caso foi configurada para executar no modo biblioteca para a arquitetura $IA-32$ (*Intel Architecture, 32-bit*), através de um *PC* (*Personal Computer*).

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

Figura 4 – Trecho do *trait* da aplicação o componente *DMEC*.

5.2 CONFIGURAÇÃO DO TESTE/DEPURAÇÃO

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

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

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

A Figura 5 traz um exemplo do arquivo de configuração de TAP para a aplicação *DMEC*.

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

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

Figura 5 – Exemplo do arquivo de configuração do teste para a TAP.

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

5.3 DISCUSSÃO SOBRE A GRANULARIDADE NA CONFIGURAÇÃO DA EXECUÇÃO DOS TESTES

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

Quando se deseja testar uma especificação bem definida, é possível

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

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

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

6 EXPERIMENTO: EXECUÇÃO AUTOMÁTICA DE TESTE E DEPURAÇÃO DE UMA APLICAÇÃO REAL

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

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

A Figura 6 apresenta a interação entre as *threads* coordenador e trabalhadoras. O coordenador é responsável por definir o particionamento de imagem, fornecer a imagem a ser processada e retornar resultados gerados para o codificador, enquanto cada trabalhador deve calcular o custo de movimento e os vetores de movimento.

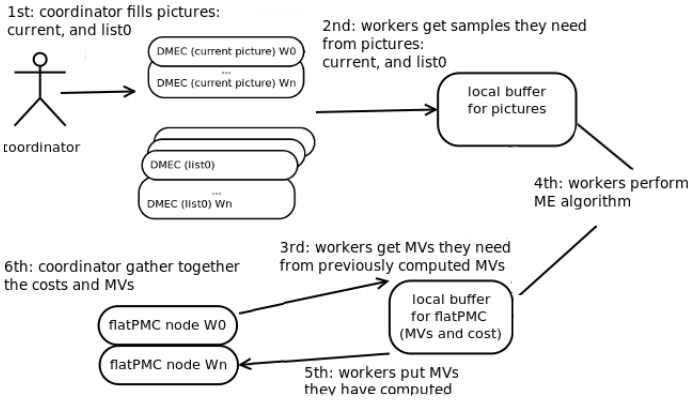

Figura 6 – Interação entre o coordenador e os trabalhadores na aplicação teste do DMEC (LUDWICH; FROHLICH, 2011).

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

trabalhadores para tentar paralelizar o trabalho da estimativa. A configuração *NUM_WORKERS* foi então testada para números entre 6 e 60. O teste do limite inferior e superior são demonstrados, respectivamente, na Figura 7 e na Figura 8.

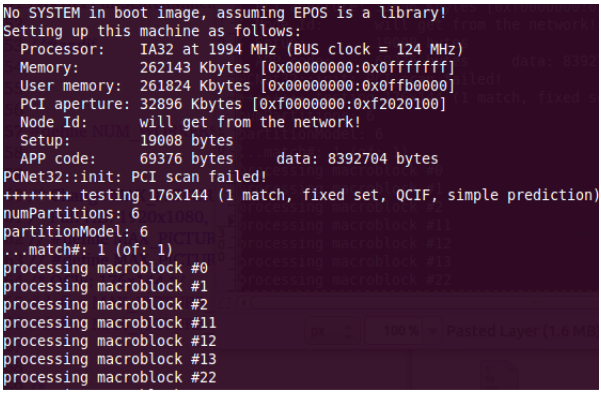

Figura 7 – Teste do DMEC com configuração `NUM_WORKERS` = 6

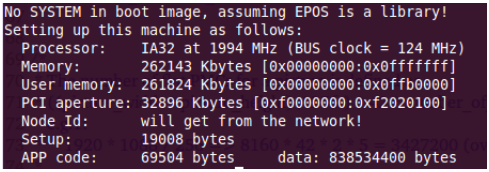

Figura 8 – Teste do DMEC com configuração `NUM_WORKERS` = 60

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

O *script* foi configurado para adicionar pontos de interrupção depois de iniciar cada uma das 5 funções da aplicação, inclusive na função principal, para descobrir se o problema encontrado era resultado de alguma delas. Foram consideradas corretas as execuções que contivessem então a resposta (*continue*) de cada uma delas. A Figura 9 mostra que não havia nenhuma resposta para a aplicação, informando que nem mesmo a função principal era atingida.

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

Figura 9 – Execução da depuração do DMEC com a configuração NUM_WORKERS = 60

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

6.1 AVALIAÇÃO DOS RESULTADOS

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

A qualidade da informação de retorno é inerente à qualidade de informação de configuração do *script* de TAP. A Figura 10 apresenta um trecho de relatório com algumas configurações geradas.

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

As Figuras 11 e 12 apresentam, respectivamente, os resultados dos experimentos relacionados à qualidade da informação devolvida para o usuário e ao consumo de tempo. Neste experimento foram realizadas 50 tentativas para cada tipo de granularidade. Para o teste parcialmente aleatório, foi modificada a propriedade NUM_WORKERS com valores em aberto e para o teste

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

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

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

determinado foi alterada esta mesma propriedade com valores de 1 a 60.

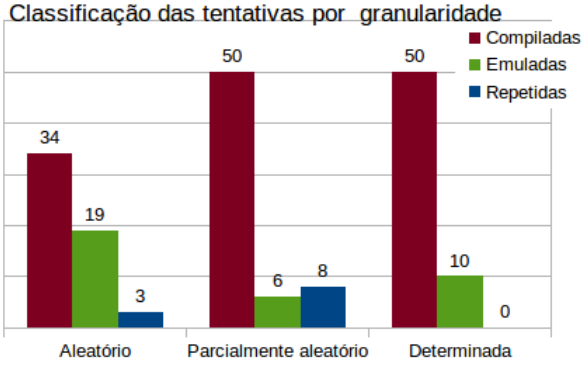

Figura 11 – Classificação das tentativas realizadas versus a configuração da granularidade.

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.

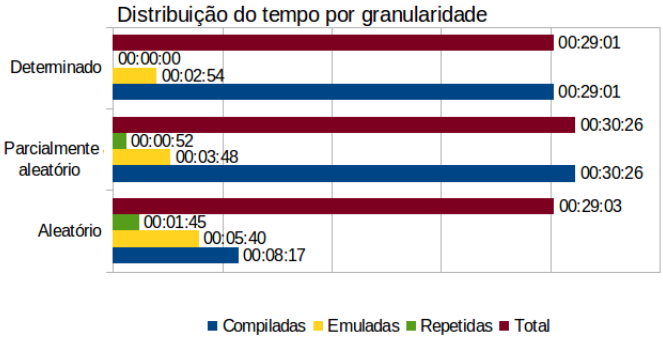

Figura 12 – Classificação das tentativas realizadas versus o consumo de tempo.

Conforme apresentado na Figura 13, a aplicação não tem uma imagem grande, mas quando adicionamos a informação extra em tempo de compilação, o consumo de memória foi aumentado em cerca de 200%. Em um sistema embarcado real, o tamanho desta nova imagem seria proibitivo.

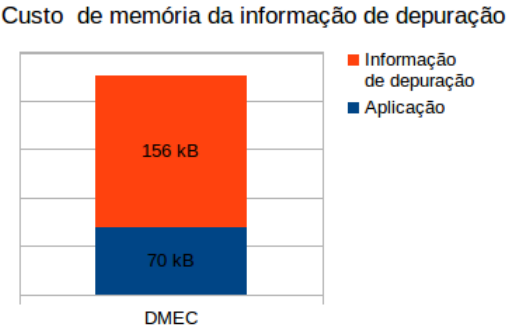

Figura 13 – Consumo de memória extra para armazenar as informações de depuração.

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

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

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

7.1 DEFINIÇÃO DAS CARACTERÍSTICAS E ANÁLISE DE *TAP*

Meta: teoria de teste universal propõe um padrão coerente de modelos/técnicas de teste, tornando possível averiguar os pontos fortes e as limitações de cada um deles e escolher racionalmente a melhor opção para cada caso.

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

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

Desafio: testes de composição Este desafio está relacionado à como testar sistemas complexos. Tradicionalmente, a complexidade de teste tem sido abordada pela decomposição do teste em ensaios

que testam separadamente alguns aspectos do sistema. Entretanto, ainda é preciso descobrir se a composição destes ensaios é equivalente ao teste do sistema todo, entender como podemos reutilizar os resultados da decomposição e quais conclusões podem ser inferidas para o sistema a partir destes resultados.

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

A Tabela 1 apresenta uma análise de TAP para os desafios propostos pela teoria de teste universal. Nela é possível observar que a ferramenta proposta atende a todos os desafios proposto, exceto o desafio de evidências empíricas, pois a modelagem ADESD possui especificidades que ainda não estão contempladas nos *testbeds* disponíveis.

Tabela 1 – Análise de TAP referente à 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 parte separada do sistema. A integração entre os componentes é apenas simulada
Evidências empíricas	A implementação e o algoritmo estão completamente liberadas para o uso de terceiros, mas devido à especificidades na modelagem do sistema **não foi possível** utilizar testes de repositórios.

Meta: modelagem baseada em teste é considerar que devemos construir o modelo para que o *software* pode ser efetivamente testado, ao invés de seguir o conceito tradicional de tomar um modelo de software e procurar alternativas para melhor explorá-lo para o teste. A idéia é migrar de teste baseado em modelos para modelagem baseada em teste.

Desafio: testes baseados em modelos são propostos para combinar os diferentes estilos de modelagem, visto que as práticas tradicionais de teste estão se tornando economicamente inviável devido ao aumento dos níveis de complexidade. É comum que sistemas complexos e heterogêneos sejam utilizados mais de um tipo de modelagem de *software* e 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 sistema em teste está em conformidade com a sua especificação, considerando alguma relação definida no modelo.

Desafio: testes baseados em anti-modelo assume que, por vezes, os modelos de *software* simplesmente não existem ou não são acessíveis. Por exemplo, em casos em que um modelo é originalmente criados, mas durante o desenvolvimento, torna-se cada vez menos úteis desde a sua correspondência com a implementação não é aplicada e se perde. Então ao invés de elaborar um plano de teste e comparar os resultados do teste com o modelo, a abordagem ani-modelo coletam informações da execução do programa e tentam sintetizar as propriedades do sistema em um modelo próximo do *software* real.

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

A Tabela **??** apresenta uma análise de todas as contribuições de TAP referente aos desafios propostos pela meta de modelagem baseada em testes. É importante ressaltar que o desafio de oráculos é uma característica relevante para a área de testes de *software*, mas não foi contemplada pela ferramente

porque foge dos objetivos específicos deste trabalho e será considerada uma melhoria futura.

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 2 – Análise de TAP referente à modelagem baseada em testes

Meta: testes 100% automáticos dependem de um ambiente de teste poderoso e que pode automaticamente providenciar a instrumentação do *software*, a geração/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.

Desafio: geração de dados de entrada para o teste sempre foi um tópico de pesquisa muito ativo, mas até hoje todos os esforços têm produzido 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.

Desafio: abordagens de teste específicas para o domínio apontam que é necessário estender abordagens específicas de domínio para a fase de testes com o intuito de encontrar métodos e ferramentas específicas de domínio e poder aprimorar a automação de teste. Neste sentido, alguns trabalhos já conseguiram demonstar a extração automática de requisitos para o teste a partir de um modelo escrito em uma linguagem fortemente tipada e específica de um domínio.

Desafio: testes *online* representam a idéia de monitorar o comportamento de um sistema em funcionamento na vida real. 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.

A Tabela **??** mostra a relação de TAP com a meta de automação completa da execução de testes de *software*. É possível notar que TAP oferece algum nível de automação em todos o desafios propostos.Entretanto, para ser cosiderada uma ferramenta 100% automática ainda serão necessários avanços na informação de dados de entrada de teste, seja a partir da extração de valores do modelo ou através de algum tipo de inteligência atificial.

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 e somente depois TAP realiza a retroalimentação de acordo com a configuração dos testes. No caso do teste não ser determinado pelo usuário, a ferramenta inclui dados iniciais aleatórios, sem adição de inteligência.
Abordagens específicas para o domínio	TAP consegue importar configurações específicas de domino e realizar testes sobe elas, contemplando de forma **parcial** a extração de dados relacionados ao domínio da aplicação.
Testes *online*	Este diferencial é apesentado pela ferramenta através do ambiente de teste e depuração, com suporte **total** aos testes durante à execução da aplicação.

Tabela 3 – Análise de TAP referente à automação total da atividade de testes de *software*

7.2 ANÁLISE COMPARATIVA

Após a contextualização das características importantes para a evolução do estado da arte e da análise qualitativa de TAP, agora as ferramentas e técnicas dos trabalhos relacionados também serão analisadas sob o mesmo ponto de vista.

A Tabela 4 apresenta uma análise comparativa entre TAP e as ferrametas correlatas no quesito teoria de teste universal.

A Tabela 5 apresenta uma análise comparativa entre as ferrametas e técnicas correlatas para os desafios propostos pela meta de modelagem baseada em teste.

Tabela 4 – Comparativo qualitativo do suporte para a meta de teoria de teste universal

Ferramentas	**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
Statistical debugging	Parcial	Sim	Parcial	Não
Program slicing	Parcial	Sim	Sim	Não
Capture and replay	Não	Parcial	Parcial	Não

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

Ferramentas	**Suporte para o desafio**		
	Teste baseado em modelos	**Teste baseado em anti-modelos**	**Oráculos de teste**
TAP	Parcial	Sim	Não
Justitia	Sim	Não	Parcial
ATEMES	Sim	Sim	Sim
Statistical debugging	Sim	Sim	Não
Program slicing	Sim	Sim	Não
Capture and replay	Sim	Sim	Parcial

A Tabela 6 apresenta uma análise comparativa entre as ferrametas e técnicas correlatas para os desafios propostos através de testes 100% automatizados.

Tabela 6 – 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	Parcial	Sim
ATEMES	Sim	Não	Sim
Statistical debugging	Não	Sim	Sim
Program slicing	Não	Sim	Sim
Capture and replay	Não	Sim	Sim

8 CONCLUSÕES

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

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

O experimento realizado apontou valores quantitativos do tempo consumido para realizar os teses e da efetividade dos ensaios. Foi confirmada que a eficácia do algoritmo está intimamente ligada à eficácia da configuração dos valores e da granularidade apresentadas à ferramenta.

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

Também foi realizada a análise qualitativa da ferramenta, levando-se em conta os desafios ainda presentes no teste de *software* e metas que necessitam ser atingidas. Os resultados demonstram o comprometimento da ferrameta com o avanço do estado da arte, pois procura oferecer uma solução viável a pelo menos 80% dos desafios propostos. Também possui resultados promissores se comparada com outras ferramentas correlatas, frequentemente apresentando soluções equivalentes às de outras ferramentas e, eventualmente, cobrindo características que não fazem parte do escopo inicial deste trabalho.

8.1 PERSPECTIVAS FUTURAS

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

- Analisar a possibilidade de reduzir o custo de memória e de tempo utiliado 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 um número de restrições eque apresentam testes mais complexos.
- Aprimorar a ferramenta para conseguir atingir todos os desafios identificados na área de teste e depuração de *software*.

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

REFERÊNCIAS BIBLIOGRÁFICAS

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

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

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

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

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

BINKLEY, D.; GOLD, N.; HARMAN, M. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 16, n. 2, p. 8, 2007.

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

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

CHEBARO, O. et al. Program slicing enhances a verification technique combining static and dynamic analysis. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2012. (SAC '12), p. 1284–1291. ISBN 978-1-4503-0857-1. <http://doi.acm.org/10.1145/2245276.2231980>.

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

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

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

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

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

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

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.

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>.

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

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

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

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

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.

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

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

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

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

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>.

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

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

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

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.

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.; 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.