TAP - Teste e Depuração de Software Embarcado

Rita de Cássia Cazu Soldi e Antônio Augusto Medeiros Fröhlich
Universidade Federal de Santa Catarina (UFSC)
Florianópolis, SC, Brasil
rita,guto@lisha.ufsc.br

***Resumo*—The process of testing and debugging embedded software is non-trivial, once it needs a thorough inspection of the entire source code to make sure that there is no behavior beyond expectations. Perform these activities on embedded software is even more defiant, once developers need to find out how to optimize the use of the scarce resources since the test itself will compete with the application under test by the scarce system resources. This paper presents `TAP`, a tool for helping developers in the process of testing and debugging embedded systems. The main idea of this tool is emulating various possible system configurations to try to find errors in the application. Once detected an unspecified behavior, `TAP` automatically perform compilation, emulation and debugging accordingly to a XML specification file.**

I. Introdução

Sempre que um desenvolvedor começa a escrever um novo código fonte, existem especificações e comportamentos que o *software* deve apresentar. Todo este processo deve ser acompanhado do teste, cujo objetivo é operar o sistema sob condições específicas e avaliar se resultados correspondem aos requisitos esperados. O teste de *software* é importante para aumentar a qualidade do produto e em alguns casos chegam a representar mais de 50% do custo do desenvolvimento [2].

Quando o teste encontra algum comportamento atípico (não especificado), deve-se depurar o *software* para procurar a causa do erro e corrigi-lo. A depuração é uma das atividades mais morosas do desenvolvimento de um *software*, uma vez que encontrar a razão para um comportamento inesperado e resolver o problema não é um processo trivial [9].

Erros em *software* podem causar problemas para as empresas envolvidas. A economia norte americana, por exemplo, pagou cerca de 60 bilhões de dólares devido a *bugs* encontrados tardiamente, sendo que este preço poderia ser reduzido para 22,5 bilhões de dólares caso fossem utilizadas técnicas mais consistentes para os testes [16].

Perdas materiais causam um grande transtorno, mas colocar em risco o meio ambiente e a vida de seres vivos é um risco inaceitável. Sendo assim, é imprescindível ter certeza do bom funcionamento de sistemas que influenciam diretamente na vida das pessoas, como por exemplo os sistemas embarcados.

Teste de *software* embarcado é uma atividade bastante desafiadora, uma vez que os recursos deste tipo de sistema são escassos e o teste estará competindo com a própria aplicação sob teste por estes recursos. Além disso, esta atividade precisa ser remodelada para cada plataforma, que depende de sistemas operacionais, arquitetura, fornecedores, ferramenta de depuração, entre outros [13]. Toda esta variabilidade faz com que os sistemas embarcados estejam mais susceptível a erros e falhas de especificação.

A automação de todo o processo de teste e depuração sem qualquer intervenção humana ainda é um desafio para os pesquisadores, embora existam alguns estudos que podem automatizar grande parte do processo a partir de dados obtidos diretamente da aplicação sob teste [6], [18].

A principal contribuição deste artigo é portanto, uma ferramenta de automação de teste e depuração de *software* embarcado, a troca automática de parâmetros de configuração (`TAP`). A `TAP` consegue capturar todas as informações necessárias a partir de uma especificação em XML, gerando assim variações dos parâmetros de configurações do software sob teste. A ferramenta também suporta a depuração e emulação da aplicação, oferecendo um relatório dos testes realizados para que o desenvolvedor consiga corrigir o problema.

II. TAP - Troca Automática de Parâmetros de Software

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

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

O algoritmo de `TAP` 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 [15].

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

Algoritmo 1: Algoritmo de Troca dos Parâmetros de Configuração

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

A implementação atual utiliza o sistema operacional (EPOS) [4], 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.

A. 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 1 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 1. Trecho do *trait* da aplicação o componente DMEC.

B. Ambiente Compartilhado de Teste e Depuração

O ambiente utilizado por TAP conta com um emulador para simular a execução da aplicação e rodar os testes. No caso de algum teste falhar, automaticamente uma ferramenta de depuração é iniciada e o resultado de todas as operações é passada para o usuário.

As ferramentas utilizadas neste ambiente podem ser substituídas por qualquer outra equivalente. A configuração será apresentada apenas para que o experimento possa ser reproduzido. Nele foi utilizado o QEMU para emular a máquina com a aplicação alvo a partir de outra máquina, usando tradução dinâmica. Desta forma torna-se possível utilizar um computador pessoal para testar aplicações compiladas para algum outro sistema embarcado. Para dar suporte também à depuração, foi necessário procurar um aliado para ver quais os passos do programa foi executado um momento antes de um *crash*. O GBD - o GNU Project Debugger se encaixou no papel, pois nele é possível especificar qualquer regra que possa afetar seu comportamento de maneira estática.

A integração de ambos é particular para cada máquina anfitriã e alvo. A Figura 2 apresenta as atividades necessárias para executar a depuração remota usando a arquitetura IA-32 como máquina anfitriã.

Primeiramente deve-se compilar com informações de depuração, depois emular a aplicação na arquitetura alvo

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

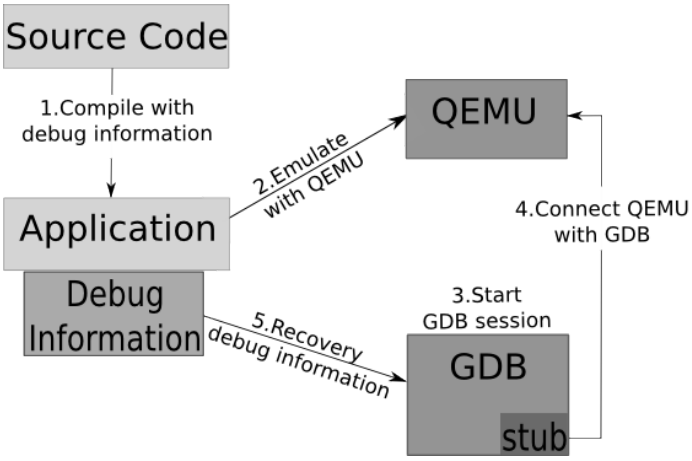

Figura 2. Integração entre QEMU e GDB.

correta. Os testes são realizados e caso algum deles falhe, o depurador deve ser iniciado. Se necessário, as informações de depuração podem ser recuperadas para ajudar a encontrar o erro. A partir desta etapa, o desenvolvedor pode definir *breakpoints, watchpoints* e até mesmo controlar a execução do programa. Todas estas atividades podem ser realizadas manualmente ou incorporadas pela TAP através do arquivo de configuração do teste.

C. 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, 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 3 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 3. 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*".

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.

III. Teste/depuração de uma aplicação real com TAP

A troca automática de parâmetros foi utilizada para testar e depurar o componente de estimativa de movimento para codificação H.264, o *Distributed Motion Estimation Component* (DMEC). Este componente executa uma estimativa de movimento explorando a semelhança entre imagens adjacentes numa sequência de vídeo que permite que as imagens sejam codificadas diferencialmente, aumentando a taxa de compressão da sequência de *bits* gerada. Estimativa de movimento é uma fase importante para codificação H.264 já que consome cerca de 90% do tempo total do processo de codificação [7].

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 4 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.

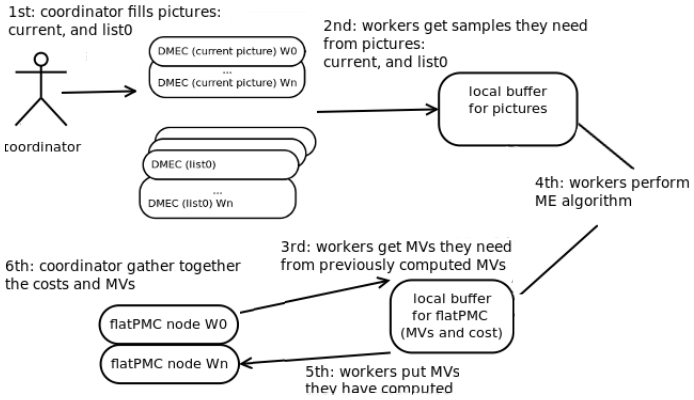

Figura 4. Interação entre o coordenador e os trabalhadores na aplicação teste do DMEC [7].

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 5 e na Figura 6.

```
No SYSTEM in boot image, assuming EPOS is a library!
Setting up this machine as follows:
  Processor:    IA32 at 1994 MHz (BUS clock = 124 MHz)
  Memory:       262143 Kbytes [0x00000000:0x0fffffff]
  User memory:  261824 Kbytes [0x00000000:0x0ffb0000]
  PCI aperture: 32896 Kbytes [0xf0000000:0xf2020100]
  Node Id:      will get from the network!
  Setup:        19008 bytes
  APP code:     69376 bytes     data: 8392704 bytes
PCNet32::init: PCI scan failed!
++++++++ testing 176x144 (1 match, fixed set, QCIF, simple prediction)
numPartitions: 6
partitionModel: 6
...match#: 1 (of: 1)
processing macroblock #0
processing macroblock #1
processing macroblock #2
processing macroblock #11
processing macroblock #12
processing macroblock #13
processing macroblock #22
```

Figura 5. Teste do DMEC com configuração `NUM_WORKERS` = 6

```
No SYSTEM in boot image, assuming EPOS is a library!
Setting up this machine as follows:
  Processor:    IA32 at 1994 MHz (BUS clock = 124 MHz)
  Memory:       262143 Kbytes [0x00000000:0x0fffffff]
  User memory:  261824 Kbytes [0x00000000:0x0ffb0000]
  PCI aperture: 32896 Kbytes [0xf0000000:0xf2020100]
  Node Id:      will get from the network!
  Setup:        19008 bytes
  APP code:     69504 bytes     data: 838534400 bytes
```

Figura 6. 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 7 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 7. DMEC debug with GDB execution with `NUM_WORKERS` = 60

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

IV. 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 8 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 9 e 10 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

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

aberto e para o teste determinado foi alterada esta mesma propriedade com valores de 1 a 60.

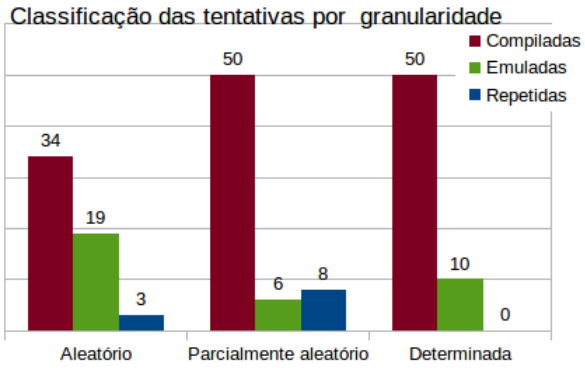

Figura 9. Classificação das tentativas realizadas versus a configuração da granularidade.

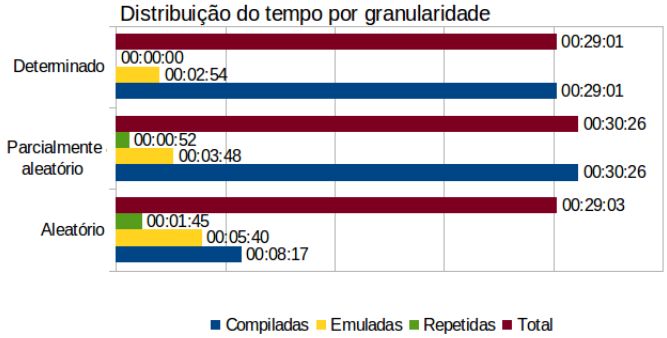

Figura 10. Classificação das tentativas realizadas versus o consumo de tempo.

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

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

Conforme apresentado na Figura 11, 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.

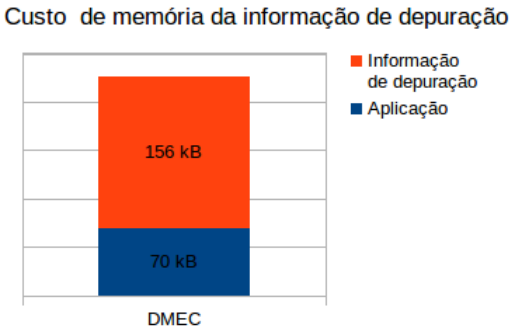

Figura 11. Consumo de memória extra para armazenar as informações de depuração.

V. Trabalhos Relacionados

A área de automação de testes possui uma vasta literatura de apoio, sendo a maioria relacionada a sistemas de propósito geral. Tanto o *script* de troca de parâmetros quando o ambiente integrado de teste e depuração tiveram seu projeto inspirado nela.

Seo et al. [14] propuseram uma ferramenta que gera e executa casos de teste para sistemas embarcados. Sua técnica que consegue identificar e classificar as interfaces das camadas do sistema embarcado, gerando testes específicos para cada uma delas. A similaridade deste trabalho com `TAP` encontra-se na maneira em que é realizada a depuração e do foco utilizado na execução dos testes, uma vez que podemos formar um paralelo entre as interfaces das camadas e as interfaces dos componentes do `EPOS`. Entretanto a vantagem de `TAP` está em utilizar o mesmo ambiente para o teste e a depuração sem precisar de intervenção manual.

`ATEMES` [5] é uma ferramenta para automação de teste para sistemas embarcados. 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 geração de dados de entrada para sistemas de múltiplos núcleos. `ATEMES` é semelhante ao presente trabalho por também executar automaticamente os testes aleatórios em ambiente de depuração cruzada para *software* embarcado. No entanto, além do teste e depuração, o ambiente integrado da `TAP` permite também a busca de pontos de melhoria na configuração da aplicação.

Trabalhos com a técnica de *statistical debugging* [20], [19], [10] utilizam dados estatísticos relacionados a várias execuções do sistema para isolar um *bug*. Esta análise estatística pode reduzir as possíveis origens de um erro apontando um *ranking* de desconfiança e identificando qual parte

do sistema gerou o erro. Devido ao grande volume de dados necessários para guardar todas as execuções e realizar a análise, esta técnica não poderia ser utilizada em um sistema embarcado real. Entretanto, o *ranking* proposto pode facilitar o trabalho do desenvolvedor, assim o projeto do ambiente da TAP foi desenvolvido para que seja possível incorporar esta técnica para melhorar o relatório fornecido ao desenvolvedor.

Em *program slicing* [12], [17], [1] a ideia principal é a partição do código e a remoção de estados ou caminhos que não levam ao erro. Esta técnica possui duas abordagens: estática e dinâmica. De um lado, a partição estática é mais rápida e aponta apenas uma aproximação do conjunto final de caminhos que podem levar ao erro. Do outro existe a partição dinâmica, que considera o conjunto de entrada inicial para decidir como fatiar o código, fornecendo uma maior precisão no conjunto de saída. Esta técnica é interessante porque necessita apenas de uma execução errada do sistema para simplificar o grupo de entradas a serem examinadas. A TAP suporta ambas abordagens através dos *traits* que podem modificar o sistema inteiro ou apenas uma parte da aplicação.

Em trabalhos que utilizam *capture and replay* [3], [11], [8] a ideia é capturar toda a execução do programa até o final e armazenar as operações envolvidas em um *log*. Burger e Zeller 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*. A proposta de TAP é executar teste e depuração em um sistema operacional no qual um componente possui uma interface que pode ser implementada de diferentes maneiras, tornando inviável manter um *log* diferente para cada implementação.

VI. Conclusão

Introduzimos neste artigo a troca automática de parâmetros de configuração (TAP) e mostramos como criar 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, para o desenvolvimento e teste. Além disso, os desenvolvedores não precisam gastar tempo compreendendo uma nova plataforma de desenvolvimento sempre que alguma característica de sistemas embarcados mudar. Este é um passo importante, pois alguns sistemas embarcados podem não ser capazes de armazenar os dados adicionais necessários para apoiar a depuração.

Foi confirmada que a eficácia do algoritmo está intimamente ligada à eficácia da configuração recebida quando TAP testou e depurou o DMEC com as três granularidades da ferramenta. A avaliação do impacto da informação de depuração no tamanho do código da aplicação foi de mais de 500% e a sobrecarga para o tempo de execução é de cerca de 60%.

A ferramenta suporta testes unitários e testes de tipagem, mas os resultados encontrados ainda podem ser melhorados em trabalhos futuros. Além de ampliar a variedade de testes, uma possível melhoria é o uso de retroalimentação do relatório de retorno de TAP.

Referências

[1] C. Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(3):223–246, 2011.

[2] A. Bertolino and A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of the Future of Software Engineering at ICSE 2007*, pages 85–103. IEEE-CS Press, 2007.

[3] M. Burger and A. Zeller. Replaying and isolating failing multi-object interactions. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ISSTA 2008*, pages 71–77. ACM, 2008.

[4] A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[5] C.-S. Koong, C. Shih, P.-A. Hsiung, H.-J. Lai, C.-H. Chang, W. C. Chu, N.-L. Hsueh, and C.-T. Yang. Automatic testing environment for multicore embedded software-atemes. *J. Syst. Softw.*, 85(1):43–60, Jan. 2012.

[6] E. Larson and R. Palting. Mdat: a multithreading debugging and testing tool. In *Proceeding of the 44th ACM technical symposium on Computer science education*, SIGCSE '13, pages 403–408, New York, NY, USA, 2013. ACM.

[7] M. Ludwich and A. Frohlich. Interfacing hardware devices to embedded java. In *Computing System Engineering (SBESC), 2011 Brazilian Symposium on*, pages 176 –181, nov. 2011.

[8] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[9] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209. ACM, 2011.

[10] S. Parsa, M. Asadi-Aghbolaghi, and M. Vahidi-Asl. Statistical debugging using a hierarchical model of correlated predicates. *Artificial Intelligence and Computational Intelligence*, pages 251–256, 2011.

[11] D. Qi, M. Ngo, T. Sun, and A. Roychoudhury. Locating failure-inducing environment changes. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 29–36. ACM, 2011.

[12] N. Sasirekha, A. Robert, and D. Hemalatha. Program slicing techniques and its applications. *Arxiv preprint arXiv:1108.1352*, 2011.

[13] S. Schneider and L. Fraleigh. The ten secrets of embedded debugging. *Embedded Systems Programming*, 17:21–32, 2004.

[14] J. Seo, A. Sung, B. Choi, and S. Kang. Automating embedded software testing on an emulated target board. In H. Zhu, W. E. Wong, and A. M. Paradkar, editors, *AST*, pages 44–50. IEEE, 2007.

[15] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[16] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, pages 02–3, 2002.

[17] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

[18] C. Zhang, J. Yang, D. Yan, S. Yang, and Y. Chen. Automated breakpoint generation for debugging. *Journal of Software*, 8(3), 2013.

[19] Z. Zhang, W. Chan, T. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 43–52. ACM, 2009.

[20] A. Zheng, M. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112. ACM, 2006.