View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoteca-e

Ministério da Agricultura, Pecuária e Abastecimento

Boletim de Pesquisa e Desenvolvimento 5

Outubro, 2002

ISSN 1677-9266

Processo de Depuração depois do Teste: Definição e Análise

depuração

Processo de depuração

depois do Teste

teste

Definição e Análise

análise

Embrapa

República Federativa do Brasil

Fernando Henrique Cardoso
Presidente

Ministério da Agricultura, Pecuária e Abastecimento

Marcus Vinicius Pratini de Moraes
Ministro

Empresa Brasileira de Pesquisa Agropecuária - Embrapa

Conselho de Administração

Márcio Fortes de Almeida
Presidente

Alberto Duque Portugal
Vice-Presidente

Dietrich Gerhard Quast
José Honório Accarini
Sérgio Fausto
Urbano Campos Ribeiral
Membros

Diretoria Executiva da Embrapa

Alberto Duque Portugal
Diretor-Presidente

Bonifácio Hideyuki Nakasu
Dante Daniel Giacomelli Scolari
José Roberto Rodrigues Peres
Diretores-Executivos

Embrapa Informática Agropecuária

José Gilberto Jardine
Chefe-Geral

Tércia Zavaglia Torres
Chefe-Adjunto de Administração

Kleber Xavier Sampaio de Souza
Chefe-Adjunto de Pesquisa e Desenvolvimento

Álvaro Seixas Neto
Supervisor da Área de Comunicação e Negócios

Embrapa

ISSN 1677-9266
Outubro, 2002

Empresa Brasileira de Pesquisa Agropecuária
Embrapa Informática Agropecuária
Ministério da Agricultura, Pecuária e Abastecimento

Boletim de Pesquisa e Desenvolvimento 5

Processo de Depuração depois do Teste: Definição e Análise

Marcos Lordello Chaim
José Carlos Maldonado
Mario Jino

Campinas, SP
2002

Embrapa Informática Agropecuária
Área de Comunicação e Negócios (ACN)
Av. André Tosello, 209
Cidade Universitária "Zeferino Vaz" – Barão Geraldo
Caixa Postal 6041
13083-970 – Campinas, SP
Telefone (19) 3789-5743 - Fax (19) 3289-9594
URL: http://www.cnptia.embrapa.br
e-mail: sac@cnptia.embrapa.br

Comitê de Publicações

Amarindo Fausto Soares
Ivanilde Dispato
José Ruy Porto de Carvalho (Presidente)
Luciana Alvim Santos Romani
Marcia Izabel Fugisawa Souza
Suzilei Almeida Carneiro

Suplentes
Adriana Delfino dos Santos
Fábio Cesar da Silva
João Francisco Gonçalves Antunes
Maria Angélica de Andrade Leite
Moacir Pedroso Júnior

Supervisor editorial: *Ivanilde Dispato*
Normalização bibliográfica: *Marcia Izabel Fugisawa Souza*
Capa: *Intermídia Produções Gráficas*
Editoração eletrônica: *Intermídia Produções Gráficas*

1ª. edição
on-line - 2002

Todos os direitos reservados

Chaim, Marcos Lordello.
Processo de depuração depois do teste: definição e análsise / Marcos Lordello Chaim, José Carlos Maldonado, Mario Jino. — Campinas : Embrapa Informática Agropecuária, 2002.
47 p. : il. — (Boletim de Pesquisa e Desenvolvimento / Embrapa Informática Agropecuária ; 5)

ISSN 1677-9266

1. Manutenção de software. 2. Depuração de software. 3. Localização de defeitos. 4. Teste de software. 5. Qualidade de software. I. Maldonado, José Carlos. II. Jino, Mario. III. Título. IV. Série.

CDD – 005.16 (21st ed.)

© Embrapa 2002

Sumário

Resumo 5

Abstract 7

Introdução 9

Material e Métodos 11

Resultados e Discussão 23

Conclusões 41

Referências Bibliográficas 42

Processo de Depuração depois do Teste: Definição e Análise

Marcos Lordello Chaim[1]
José Carlos Maldonado[2]
Mario Jino[3]

Resumo

O objetivo da realização deste trabalho foi analisar em detalhes o processo de depuração que ocorre *depois* da atividade de teste. Para tanto, vários processos de depuração de software foram analisados. Baseado nessa análise, foi definido um processo de *Depuração depois do Teste* (DDT) que enfatiza os seguintes aspectos: identificação, avaliação e refinamento sucessivo de sintomas internos até a localização do defeito; e tipo de informação de teste utilizada. Várias técnicas de depuração foram então avaliadas em termos de mecanismos de apoio ao processo DDT e quanto a sua escalabilidade para programas reais. Dessa avaliação, observou-se que os resultados do teste sistemático de software, quando utilizados na depuração, resultam em técnicas de baixo custo e com maiores perspectivas de escalabilidade para programas reais. Infelizmente, esses resultados têm sido utilizados apenas para apoiar um dos aspectos do processo DDT – *identificação de sintomas internos*. Essa observação motiva a definição de estratégias de depuração que utilizem, de maneira eficaz e eficiente, as informações de teste na *avaliação* e no *refinamento de sintomas internos*.

Termos para indexação: manutenção de software, depuração de software, localização de defeitos, teste estrutural e funcional.

---

[1] Engenheiro Elétrico, Mestre e Doutor em Engenharia Elétrica, Pesquisador da Embrapa Informática Agropecuária, Caixa Postal 6041, Barão Geraldo – 13083-970 – Campinas, SP. Durante a realização deste trabalho, foi apoiado parcialmente pela Fundação CAPES.

[2] Mestre em Sistemas de Computação e Doutor em Engenharia Elétrica, Prof. do Instituto de Ciências Matemáticas e de Computação, USP, Caixa Postal 668 – 13560-970 – São Carlos, SP.

[3] Mestre em Engenharia Elétrica e Doutor em Ciências da Computação, Prof. da FEEC/Unicamp, Caixa Postal 6101 – 13083-970 – Campinas, SP.

The Debugging after Testing Process: Definition and Analysis

Abstract

Our goal in this paper was to analyze in detail the debugging process that occurs after software testing. Regarding this objective, several processes of software debugging were analyzed. Based on such analysis, a process of Debugging After Testing (DAT) was defined; this process strengthens especially the following aspects: identification, assessment and refinement of internal symptoms up to the fault localization; and the kind of testing information utilized to support the preceding activities. Several debugging techniques were then assessed regarding their support to DAT and their scalability to real-world programs. From this analysis, we have observed that debugging techniques based on testing information have lower costs and are likely to scale up. Unfortunately, such debugging techniques support a single aspect of DAT – identification of internal symptoms. This observation indicates the need of effective and efficient debugging strategies that utilize testing information to support all aspects of DAT.

Index terms: software maintenance, software debugging, fault localization, structural and functional testing.

Introdução

A atividade de depuração ocorre no processo de software em três momentos distintos: *durante a codificação*, *depois do teste* e *durante a manutenção*. Durante a codificação, a depuração é uma ferramenta complementar à programação. O cenário típico ocorre quando o programador codifica parte da especificação/projeto e prepara um teste não-sistemático para verificar o novo código. Em geral, ele executa o programa e verifica o resultado. Se incorreto, o novo código deve ser depurado. Outra possibilidade é não executar o programa de uma vez, mas passo a passo no trecho relativo ao novo código. Assim, as características da depuração durante a codificação são: 1) uso de testes não-sistemáticos; e 2) ênfase na análise do novo código introduzido, não na localização de defeitos.

A ferramenta típica utilizada neste cenário é o *depurador simbólico* (Stallman & Pesch, 1999; Adams & Muchnick, 1986). Esta ferramenta tradicional de depuração presta-se bem à *depuração durante a codificação* porque a tarefa de localização do defeito é reduzida, visto que há uma grande probabilidade do comportamento incorreto estar localizado no novo código. Embora o defeito possa também estar localizado na interface do código antigo com o novo ou mesmo no código antigo, tendo sido revelado quando o novo foi introduzido, essas duas possibilidades ocorrem com menor freqüência. Outro instrumento importante para análise do novo código são as ferramentas que detectam usos inválidos de memória. Ferramentas comerciais como Purify (Hastings & Joyce, 1992) permitem a detecção de *vazamento* de memória (*memory leaking*) e acesso inválido por uso de ponteiros.

Já depuração que ocorre *depois* da atividade de teste possui características diferentes. O objeto da depuração neste momento é o software obtido depois de completada a fase de implementação e que, supostamente, já possui todas as funções estabelecidas na especificação. Além disso, a depuração recebe como entrada não somente o código e a especificação, mas também os *resultados* da atividade de teste. Infelizmente, em ambientes industriais, as ferramentas utilizadas atualmente para a *depuração depois do teste* são as mesmas utilizadas durante a codificação.

Várias técnicas que utilizam informação de teste durante a depuração de programas têm sido pesquisadas e propostas (Korel & Laski, 1988; Fritzson et al., 1992; Chen & Cheung, 1997; DeMillo et al., 1996; Collofello & Cousins, 1987; Weiser, 1984; Lyle & Weiser, 1987; Wong et al., 1999;

Agrawal et al., 1998; Agrawal et al., 1995) e pelo menos uma ferramenta comercial resultante desses esforços está disponível (Agrawal et al., 1998). Alguns fatores, porém, têm dificultado a difusão dessas técnicas na prática, pois: 1) muitas delas utilizam informação trivial de teste (e.g., se o caso de teste revela ou não um defeito); 2) em parte em decorrência do primeiro fator, algumas técnicas possuem alto custo em *tempo de depuração*; e 3) não existem técnicas para mapear a informação de teste em informação dinâmica, observável durante a execução do programa e para refiná-la até a localização do defeito.

Durante a manutenção, novamente há a necessidade de depuração do software. Pressman (1994) lista quatro tipos diferentes de manutenção: *corretiva*, *aperfeiçoadora*, *adaptativa* e *preventiva*. Em todos os tipos de manutenção, com exceção da *corretiva*, ocorre um novo processo de desenvolvimento, implicando a codificação e teste das novas funções identificadas; portanto, ocorrem recorrentemente *depuração durante a codificação* e *depois do teste*. Durante a manutenção *corretiva*, o que essencialmente ocorre é a *depuração depois do teste*, porém, com um problema adicional: o código precisa ser *entendido*, visto que o programador original pode não estar mais disponível ou não se lembrar mais das funções implementadas. O problema de *entendimento* do programa é inerente a qualquer atividade de manutenção e requer o uso de técnicas de compreensão de programas. Já o conjunto de casos de teste disponível para depuração é definido pelo conjunto originalmente desenvolvido durante o teste mais o caso de teste que provoca a ocorrência da falha no software liberado.

Neste trabalho a atividade de depuração que ocorre *depois* do teste é investigada em mais detalhes. Em Material e Métodos são apresentadas as definições de termos relacionados às atividades de teste e depuração utilizados no decorrer do texto, a descrição das técnicas de teste cujos resultados podem ser utilizados na depuração e uma análise dos principais processos de depuração encontrados na literatura. A partir dessa análise, em Resultados e Discussão, é estabelecido o Processo de *Depuração depois do Teste* (DDT). Este processo tem como objetivo indicar onde as técnicas e as ferramentas podem, utilizando informação (resultados) de teste, melhorar o processo de depuração. Várias técnicas de depuração de programas procedimentais são então discutidas e avaliadas com relação ao processo DDT e a sua escalabilidade para sistemas reais. Em Conclusões os principais resultados obtidos e os trabalhos futuros são apresentados.

Material e Métodos

As definições descritas a seguir são baseadas no padrão IEEE número 610.12-1990 (IEEE Computer Society, 1991). No contexto desse relatório, *engano* é a ação humana que pode levar a um defeito. O *defeito*, por sua vez, é a manifestação *física* do engano em uma representação do software. Uma *falha* é a ocorrência observável de um ou mais defeitos quando o software é testado ou utilizado em campo. O estado intermediário ou final, caracterizado por um comportamento incorreto ou um desvio da especificação, é chamado de *erro*.

Dessa maneira, o processo de ocorrência de falhas pode ser resumido da seguinte forma: um *engano* (ação humana) leva a um *defeito* (deficiência algorítmica) que provoca a ocorrência de uma *falha* durante o teste. Antes da ocorrência da falha, *erros* podem ocorrer. O caso de teste é *revelador de defeito* quando uma falha ocorre ao ser executado. Se nenhuma falha ocorre, o caso de teste é então *não-revelador de defeito*.

O Programa exemplo 1 (escrito em linguagem C) contido na Fig. 1 é utilizado para exemplificar os conceitos de defeito, erro e falha. Sua especificação estabelece que os oito elementos do vetor `a` devem ser iniciados com os valores 0, 1, 2, 3, 4, 5, 6 e 7 e o *i-ésimo* elemento do vetor deve ser impresso. Durante a codificação do programa, dois enganos produziram dois defeitos. O primeiro está localizado na cadeia de caracteres do comando `printf` na linha 1; e o segundo na expressão do comando `while`. Ambos os defeitos provocam a ocorrência de falhas quando casos de teste são executados.

```
    Main()
    {
    Int i, a [ 8] , l;
1:  Printf("Enter i (0 <= i < 9): "); /* ←correto: < 8 */
2:  Scanf ("%d", &i);

3:  l = 0;
4:  While( l < 9) /*← correto: (l < 8) */
      {
5:    a [ l] = l;
6:    ++l;
      }

7:  Printf("a [ %d] = %d\n", i, a[ i] );
    }
```

Fig. 1. Programa exemplo 1.

Suponha-se que o programa tenha sido executado com o valor de entrada (`i`= 5). Para esse caso de teste, os valores de saída produzidos são (`i`= 8, `a [ i]` = 8); no entanto, as saídas esperadas são (`i`= 5, `a [ i]` = 5). A Fig. 2 descreve a execução desse caso de teste utilizando pares X^p, chamados *pontos de execução*, que indicam que o comando *X* foi o *p-ésimo* comando executado. Duas falhas ocorrem. A primeira é a saída errada no ponto de execução 1^1. A segunda manifesta-se nos valores de saída incorretos no ponto 7^{32}.

Comando X^p	Código Fonte
1^1	`printf("Enter i (0 <= i < 9): ");`
2^2	`scanf ("%d", &i);`
3^3	`l=0;`
4^4	`while(l < 9)`
5^5	`a[ l] =l;`
6^6	`l++;`
4^7	`while(l < 9)`
5^8	`a[ l] =l;`
6^9	`l++;`
4^{10}	`while(l < 9)`
...	...
4^{28}	`while(l < 9)`
5^{29}	`a[ l] =l;`
6^{30}	`l++;`
4^{31}	`while(l < 9)`
7^{32}	`Printf("a[ %d] = %d\n",i,a[ i] );`

Fig. 2. Execução do programa exemplo 1 com a entrada (i=5).

Enquanto as falhas podem ser observadas nas saídas produzidas pelo programa, os erros requerem a observação dos estados intermediários do programa. Por exemplo, no caso de teste descrito acima, dois erros ocorrem antes da ocorrência da segunda falha. A especificação estabelece que os oito elementos do vetor `a` devem receber certos valores iniciais; portanto, o corpo do comando `while`, que atribui esses valores, deve ser percorrido oito vezes. No entanto, o laço é percorrido nove vezes. Isto é um erro porque se trata de um desvio da especificação.

O segundo erro está relacionado com o estado da memória do programa depois do ponto de execução 5^{29}, apresentado na Fig. 3. Note-se que o endereço de memória Oxbffffbd4 está vinculado estaticamente à variável `i`; todavia, esta posição de memória recebe o valor 8 em 5^{29}. Isto ocorre porque, devido ao defeito contido no comando `while`, o endereço de memória Oxbffffbd4 neste ponto está também vinculado ao elemento *fictício* `a [ 8]` do vetor `a`. Trata-se de um estado intermediário incorreto do programa e, portanto, de um erro.

Valor	Variável: endereço
8	i, "a[8]": 0xbffffbd4
7	a[7]: 0xbffffbd0
6	a[6]: 0xbffffbcc
5	a[5]: 0xbffffbc8
4	a[4]: 0xbffffbc4
3	a[3]: 0xbffffbc0
2	a[2]: 0xbffffbbc
1	a[1]: 0xbffffbb8
0	a[0]: 0xbffffbb4
8	l: 0xbffffbb0

Fig. 3. Estado da memória do programa depois do ponto de execução 5^{29}.

Os erros e falhas de um caso de teste podem ser mapeados para sintomas internos. Os *sintomas internos* do programa exemplo são caracterizados pelo valor de uma ou mais variáveis em um determinado ponto de execução. Em outras palavras, os sintomas internos indicam um ponto de execução em que o erro ou a falha podem ser investigados. Por exemplo, o erro relacionado com o número incorreto de execuções do comando `while` pode ser mapeado para o valor da variável `l` (no caso, 9) no ponto de execução 4^{31}. De maneira semelhante, o valor da variável `i` (no caso, 8) no ponto de execução 5^{29} é o sintoma interno do erro associado à invasão de memória. Por sua vez, o sintoma interno

relacionado com a segunda falha observada são os valores das variáveis `i` e `a[i]` (respectivamente, 8 e 8) no último comando executado (ponto de execução 7[32]). O processo de localização do defeito começa depois do mapeamento das falhas e erros para sintomas internos.

Uma vez definidos os principais termos relacionados ao teste de software, são apresentadas a seguir as principais técnicas de teste. Um programa pode ser testado de maneira *ad hoc* ou *sistemática*. Na primeira abordagem, os casos de teste são derivados unicamente a partir da intuição do testador. Já o teste sistemático implica que os casos de teste foram derivados utilizando uma técnica de teste. Um dos objetivos das técnicas sistemáticas de teste é garantir que aspectos considerados importantes da especificação e da implementação do programa tenham sido exercitados pelo menos uma vez por algum caso de teste; elas podem ainda ter como objetivo detectar os defeitos mais comuns ou identificar máquinas de estados finitos errôneas. Por isso, as técnicas de teste são classificadas em *funcionais*, *estruturais*, *baseadas em defeitos* e *baseadas em máquinas de estados finitos*.

A técnica funcional deriva requisitos de teste a partir das especificações do software. Exemplos de técnicas funcionais são: *Particionamento de Equivalência*, *Análise de Valores Limites*, *Grafos de Causa e Efeito*, *Categorização-Particionamento*, etc. (Ostrand & Balcer, 1988; Myers, 1979). Já a técnica baseada em defeitos estabelece os requisitos de teste explorando os defeitos mais comuns. Os critérios *Análise de Mutantes* (DeMillo et al., 1978) e *Semeadura de Defeitos* (Budd, 1981) são exemplos de técnicas baseadas em defeitos. A técnica de teste com base em máquinas de estados finitos, por sua vez, utiliza a estrutura de máquinas de estado finito e o conhecimento subjacente para derivar requisitos de teste (Maldonado & Fabbri, 2001).

A seguir, as técnicas de teste estruturais e baseadas em defeitos são descritas em mais detalhes visto que seus resultados são utilizados pelas técnicas de depuração analisadas.

A técnica de teste estrutural (também chamada de técnica de teste caixa branca) requer que os casos de teste selecionados executem (exercitem) determinados caminhos do programa considerados relevantes para que o testador aumente a sua confiança em relação à corretitude do programa. Os caminhos requeridos definem requisitos de teste que constituem um critério de adequação do conjunto de casos de teste. Em outras palavras, o conjunto de casos de teste *T* é considerado adequado, do ponto de

vista do teste estrutural e de acordo com um critério C, se o conjunto de requisitos de teste (RT_C) estabelecido por C é exercitado pelos casos de teste de T.

Para avaliar o grau de adequação do conjunto T a um critério C é utilizada a medida de cobertura $MT_C(T)$ definida a seguir. Seja $RT_C(t)$ o conjunto de requisitos de teste estabelecidos pelo critério C exercitados por um caso de teste $t \in T$. A medida $MT_C(T)$ é dada pela relação a seguir:

$$MT_C(T) = (|\cup RT_C(t)| \text{ para todo } t \in T) / RT_C$$

onde $|S|$ representa o número de elementos de um conjunto S.

Os *resultados* do teste utilizando um critério estrutural C incluem os conjuntos de casos de teste T, de casos de teste reveladores de defeito $T_R \subset T$, de casos de teste não-reveladores de defeito $T_{NR} \subset T$ e de requisitos de teste RT_C e $RT_C(t)$, além da medida de cobertura $MT_C(T)$.

A seguir, alguns critérios de teste estruturais são descritos. Antes, porém, são introduzidos os conceitos básicos de fluxo de controle e de fluxo de dados, utilizados na definição dos requisitos de teste dos critérios estruturais.

Seja P um programa escrito em uma linguagem do estilo Algol (Ghezzi & Jazayery, 1987) e $S_1 \ldots S_i \ldots S_n$, $1 \leq i \leq n$, a sua seqüência de comandos. P pode ser mapeado para um grafo de fluxo de controle $G(N, R, s, e)$ onde N é o conjunto de blocos de comandos (*nós*) tal que uma vez executado o primeiro comando do bloco todos são executados seqüencialmente, e é o nó de entrada, s é o nó de saída e R é o conjunto de ramos que representam a possível transferência de fluxo de controle entre dois nós. A Fig. 4 apresenta o grafo de fluxo de controle obtido a partir do programa da Fig. 1. Um caminho é a seqüência de nós $(n_i, \ldots, n_k, n_{k+1}, \ldots, n_j)$, onde $i \leq k < j$, tal que $(n_k, n_{k+1}, \in R$. Um caminho é livre de laço se todos os nós são distintos. Um caminho é completo quando começa em e e termina em s.

Uma *definição de variável* (*def*) ocorre sempre que um valor é armazenado em uma posição de memória. A ocorrência de uma variável é um *uso* quando ela não estiver sendo definida. Dois tipos de usos são distinguidos: *c-uso* e *p-uso*. O primeiro tipo afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado. O segundo tipo afeta diretamente o fluxo de controle do programa.

Um caminho (i, n_1, ... , n_m, j), $m \geq 0$, que não contenha definição de uma variável x nos nós n_1, ... , n_m é chamado de *caminho livre de definição* com respeito a (c.r.a.) x do nó i ao nó j e do nó i ao arco (n_m, j). Um caminho livre de definição (n_1, n_2, ... , n_j, n_k) c.r.a. a uma variável x, onde o caminho (n_1, n_2, ... , n_j) é um caminho livre de laço e n_1 tem uma definição de x, é denominado potencial-du-caminho[4] c.r.a. x.

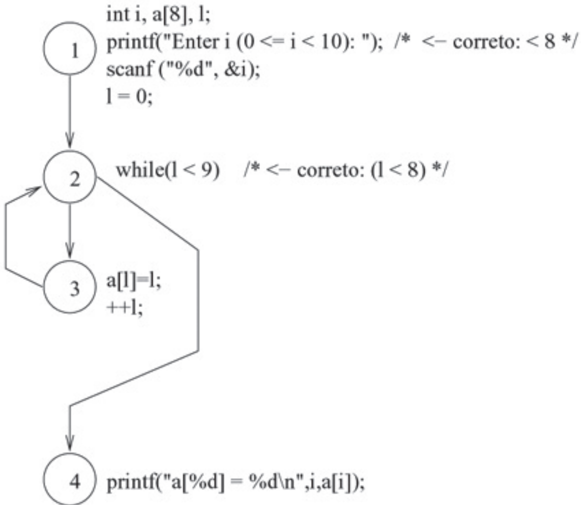

Fig. 4. Grafo de fluxo de controle obtido do programa da Fig. 1.

Os critérios baseados em fluxo de controle têm seus requisitos de teste associados a elementos do fluxo de controle do programa. Os principais critérios baseados em fluxo de controle são:

- **Critério Todos Nós:** exige que cada nó do grafo seja executado pelo menos uma vez.
- **Critério Todos Ramos:** exige que cada ramo do grafo seja executado pelo menos uma vez.
- **Critério Todos Caminhos:** exige que cada caminho, dado por uma seqüência finita de nós do grafo, seja executado pelo menos uma vez.

[4] *Potencial du-caminho* é a forma simplificada para *caminho definição-uso potencial*. A diferença entre caminho livre de definição e potencial du-caminho é que no primeiro podem ocorrer laços; no segundo, não.

O teste segundo o critério todos caminhos é chamado *ideal* ou *exaustivo* do ponto de vista estrutural. O teste exaustivo de programas seria o mais indicado, porém, na maioria das vezes, é impraticável visto que o número de caminhos é muito grande ou até mesmo infinito (quando laços estão presentes).

Os critérios baseados em análise de fluxo de dados (chamados simplesmente de critérios de fluxo de dados) (Ntafos, 1984; Rapps & Weyuker, 1985; Ural & Yang, 1988; Maldonado et al., 1992; Herman, 1976; Laski & Korel, 1983) estabelecem requisitos de teste que exigem a execução de caminhos entre a *definição* e o (potencial) *uso* de uma variável; por isso, os seus requisitos de teste são chamados de *associações definição-(potencial)-uso*.

A intuição subjacente a esses critérios é que a confiança em relação à corretitude do programa é aumentada se todo valor atribuído a uma variável for utilizado pelo menos uma vez na execução do conjunto de casos de teste (Frankl & Weyuker, 1988). Os critérios Potenciais Usos (Maldonado et al., 1992) estenderam essa intuição utilizando o conceito de *potencial uso*. Segundo este conceito, os requisitos de teste devem exigir caminhos entre uma definição e os pontos do programa onde o valor da definição *pode* ser utilizado, ou seja, onde há um potencial uso.

A seguir, são descritos alguns critérios de fluxo de dados das famílias Fluxo de Dados (Rapps & Weyuker, 1985) e Potenciais Usos (Maldonado et al., 1992):

- **Critério Todos P-usos**: requer que todas as associações definição-uso (*adu*) do tipo (i, (j,k), x) do programa em teste sejam exercitadas pelos casos de teste de um conjunto T. Uma associação (i, (j,k), x) é exercitada quando pelo menos um caminho livre de definição c.r.a. x do nó i até o ramo (j,k) é executado por um caso de teste $t \in T$.
- **Critério Todos Usos**: requer que todas as associações definição-uso dos tipos (i, j, x) e (i, (j,k), x) sejam exercitadas pelos casos de teste de um conjunto T. Uma associação (i, j, x) ou (i, (j,k), x) é exercitada quando pelo menos um caminho livre de definição c.r.a. x do nó i até o nó j ou ramo (j,k) é executado por um caso de teste $t \in T$.
- **Critério Todos Potenciais-Usos**: requer que todas as associações definição-potencial-uso (*adpu*) dos tipos (i, j, x) e (i, (j,k), x) sejam exercitadas pelos casos de teste de um conjunto T. Note-se que

para caracterizar uma associação definição-potencial-uso não é necessário um uso explícito de *x* em *j* ou (*j*,*k*), apenas que o nó *j* ou ramo (*j*,*k*) seja alcançável por um caminho livre de definição c.r.a. *x* a partir de *i*.

- **Critério Todos Potenciais-Usos/Du**: também requer que todas as associações definição-potencial-uso dos tipos (*i, j, x*) e (*i*, (*j*,*k*), *x*) sejam exercitadas pelos casos de teste de um conjunto *T*, porém, por potenciais du-caminhos.

Várias ferramentas foram desenvolvidas para apoiar a utilização dos critérios definidos acima: as ferramentas ASSET (Frankl et al., 1985), PROTESTE+ (Silva, 1995) e APODOS (Marx & Frankl, 1999, 1996) apóiam a aplicação dos critérios de Rapps e Weyuker para a linguagem Pascal; ATAC (Horgan & London, 1991) e Tactic (Ostrand & Weyuker, 1991) são ferramentas que apóiam a aplicação dos critérios de Rapps e Weyuker para a linguagem C; e a ferramenta POKE-TOOL (Chaim et al., 1998) apóia a aplicação dos critérios todos nós, todos ramos e das famílias Fluxo de Dados e Potenciais Usos para várias linguagens de programação (C, FORTRAN, COBOL).

As técnicas *baseadas em defeitos* derivam casos de teste a partir de defeitos específicos (ou classes de defeitos) comuns em linguagens de programação. O objetivo é mostrar a presença ou ausência de tais defeitos no programa. O principal critério baseado em defeitos é a *análise de mutantes*.

A idéia básica do critério de *análise de mutantes* foi apresentada por DeMillo et al. (1978) e é conhecida como hipótese do *programador competente*. Esta hipótese assume que os programadores experientes escrevem programas muito próximos do correto. Os programas incorretos contêm um desvio sintático que leva a resultados errados.

A análise de mutantes visa à seleção de casos de teste que sejam capazes de identificar a presença dos possíveis desvios sintáticos (defeitos) mais comuns. Para determinar um conjunto de casos de teste com esta propriedade, programas mutantes são gerados através da aplicação de *operadores de mutação*. Os operadores são regras que são aplicadas para definir alterações no programa em teste. Por exemplo, o operador *eliminação de comando* gera um programa mutante em que um comando do programa original foi eliminado. Casos de teste devem então ser gerados para diferenciar o comportamento do programa original do

comportamento dos programas mutantes. Quando existir essa diferença, o mutante é dito estar *morto*. Se a saída do programa original está correta, então ele está livre do possível defeito contido no programa mutante. O critério de análise de mutantes exige que todos os mutantes sejam mortos.

Analogamente aos critérios de teste estrutural, uma medida do grau de cobertura (chamada *score* de mutação) do conjunto de casos de teste T em relação ao critério de análise de mutantes através de um operador de mutação θ é definido. Seja M_θ o conjunto de programas mutantes M obtidos através da aplicação do operador θ em um programa P e $M_\theta(t)$ o conjunto de mutantes M que são mortos por um caso de teste $t \in T$. O *score de mutação* (S_θ) do conjunto T em relação ao critério de análise de mutantes através do operador θ é dado pela relação abaixo:

$$S_\theta(T) = (|\cup M_\theta(t)| \text{ para todo } t \in T) / M_\theta$$

Os *resultados* do teste de mutantes através do operador θ incluem os conjuntos de casos de teste T, de casos de teste reveladores de defeito $T_R \subset T$, de casos de teste não-reveladores de defeito $T_{NR} \subset T$ e de mutantes M_θ e $M_\theta(t)$, bem como o *score* de mutação $S_\theta(T)$.

A seguir inicia-se a análise dos principais processos de depuração definidos na literatura. O processo *Hipótese-Validação* é baseado na elaboração e validação interativa de hipóteses (Araki et al., 1991; Vessey, 1985). De acordo com este processo, o mantenedor — entendido como o responsável pela depuração do software — estabelece hipóteses com relação à localização do defeito e à modificação necessária para corrigir o programa. A depuração é guiada pela verificação e refutação das hipóteses levantadas, bem como pela geração de novas hipóteses e refinamento das já existentes. O algoritmo abaixo resume o processo:

1. Elaborar um conjunto inicial de hipóteses a partir do relatório de teste;
2. Modificar o conjunto de hipóteses;
3. Selecionar uma hipótese;
4. Verificar hipótese selecionada;
5. Verificar se o defeito foi encontrado; se sim, dirigir-se ao passo 6; caso contrário, retornar ao passo 2;
6. Elaborar um conjunto inicial de hipóteses para modificação;

7. Modificar o conjunto de hipóteses;
8. Selecionar uma hipótese;
9. Verificar hipótese selecionada;
10. Verificar se o defeito foi removido; se sim, o processo de depuração está terminado; caso contrário, retornar ao passo 7.

As hipóteses podem ser derivadas a partir das falhas observadas, do código fonte, da especificação, do comportamento esperado e real do programa, etc. A *seleção* e *verificação* de hipóteses requerem muitas vezes a *simplificação* da condição para a ocorrência da falha, a *redução* do espaço de busca e a *avaliação* de hipóteses.

O *Processo Sistemático de Depuração* foi desenvolvido por Agrawal (1991) e posteriormente expandido por DeMillo et al. (1996) e Pan (1993). Agrawal propôs uma interpretação do processo Hipótese-Validação baseado em duas técnicas — *execução em reverso*[5] e *slicing dinâmico de programas*[6] – cujo objetivo é automatizar as atividades do processo proposto. O processo é descrito a seguir:

1. Detectar falha;
2. Escolher critério de *slicing*;
3. Obter *slicing dinâmico*;
4. Sugerir localização do defeito;
5. Restabelecer o estado do programa na localização sugerida;
6. Examinar estado do programa reestabelecido para confirmar localização do defeito;
7. Se o defeito foi localizado, então o processo de localização está terminado; caso contrário, o mantenedor pode escolher ir para os passos 2, 4 ou 6 e continuar o processo de localização do defeito.

---

[5] Alguns depuradores simbólicos permitem que o programa seja executado em sentido contrário, recuperando seu estado anterior sem a necessidade de reexecutá-lo (Boothe, 2000; Agrawal et al., 1991; Lieberman & Fry, 1998; Balzer, 1969; Tolmach & Appel, 1995). Estas ferramentas serão discutidas em Resultados e Discussão.

[6] A técnica de fatiamento dinâmico seleciona comandos do programa que afetam um determinado critério de fatiamento (conjunto de variáveis em um determinado ponto de execução do programa) para uma entrada em particular (Korel & Laski, 1988; Agrawal & Horgan, 1990). Esta técnica é descrita em mais detalhe em Resultados e Discussão.

Este processo requer a eleição de um critério de fatiamento pelo mantenedor. Esta tarefa exige que a falha observada seja mapeada para um sintoma interno, que pode ser o valor de uma variável de saída incorreto ou o valor incorreto de uma variável intermediária em um determinado ponto da execução do programa. No caso da variável de saída, o ponto de execução é o último comando executado. O mapeamento estabelece o critério de fatiamento (variável, ponto de execução) que permite a aplicação da técnica de fatiamento dinâmico para reduzir o espaço de busca do defeito. A partir de um espaço de busca reduzido, o mantenedor deve sugerir uma possível localização do defeito. A execução em reverso permite que o mantenedor acesse o estado do programa na localização suspeita e verifique se o defeito está localizado nela.

O Processo de Depuração de Pan (DeMillo et al., 1996; Pan, 1993) enfatiza a conexão entre as atividades de teste e depuração, e vice-versa, estendendo o processo sistemático de Agrawal. Os autores consideram a existência de uma ferramenta integrada de teste e depuração para definir o processo que se segue:

1. O usuário (da ferramenta) observa e analisa falhas depois do teste completo do programa antes de mudar para o modo de depuração;
2. A ferramenta interativamente auxilia o usuário a reduzir o espaço de busca utilizando instrumentação dinâmica (e.g. fatiamento dinâmico e execução em reverso) e informação de teste. Neste estágio os usuários podem facilmente chavear entre o modo de teste e depuração;
3. O usuário deve realizar análises adicionais baseadas no espaço de busca reduzido e informação de teste para localizar os defeitos;
4. Depois dos defeitos terem sido localizados e corrigidos, o usuário pode retestar o programa para certificar-se de que os defeitos foram eliminados.

A diferença fundamental deste processo em relação ao processo sistemático é a ênfase no uso de informação de teste. O primeiro passo do processo refere-se ao mapeamento das falhas em sintomas internos; o segundo preconiza a redução do espaço de busca não somente utilizando fatiamento dinâmico e execução em reverso, mas também informação de teste. O terceiro passo refere-se ao processo de

refinamento da busca do defeito, sugerido no processo sistemático de Agrawal pelo laço contido no passo 7. Para terminar, o processo enfatiza a necessidade do teste de regressão, o que é facilitado em um ambiente em que teste e depuração estão integrados.

O Processo de Depuração de Chan (Chan, 1997) é, como os demais, baseado no processo Hipótese-Validação. O autor argumenta que o mantenedor, para criar e validar uma hipótese, deve realizar uma das seguintes tarefas, possivelmente mais de uma vez:

1. Executar o programa com o dado que provoca a manifestação da falha;
2. Inspecionar o dado de entrada e de saída e o seu relacionamento;
3. Inspecionar os valores das variáveis em estados intermediários do programa;
4. Inspecionar os comandos que foram executados.

O problema segundo Chan é quando tanto o número de estados intermediários quanto o número de comandos suspeitos é grande. Esta situação é chamada de explosão de informação na depuração. Para resolver este problema, é proposto um processo baseado em duas técnicas: *fatiamento de dados* e *fatiamento dinâmico* de programas. O conceito de *fatia de dados* é introduzido por Chan & Lakhotia (1998) e definido a seguir: seja d_1 um dado de entrada que provoca a manifestação de uma falha em um programa p, o dado de entrada d_2 é um fatia de dados de d_1 se:

1. d_2 reproduz em p a mesma falha que d_1; e
2. d_2 é menor que d_1, isto é, a soma das informações relativas aos estados intermediários de d_2 é menor do que a de d_1 durante a execução de p.

Técnicas para determinação de *fatias* de dados foram desenvolvidas por Chan & Lakhotia (1998). Para diminuir o número de comandos suspeitos relacionados com a falha observada, o processo de Chan preconiza o uso de *fatiamento* dinâmico de programas. Em particular, é requerido o uso de *fatias* dinâmicas executáveis uma vez que elas podem ser executadas com as *fatias* de dados. Assim, o problema de *explosão de informação* na depuração é enfrentado com dados e programas *menores*. O processo proposto por Chan é, portanto, orientado para as situações em que é necessária a simplificação das condições para a manifestação da falha.

O processo Hipótese-Validação é um *modelo cognitivo* que descreve a maneira como os seres humanos abordam a identificação e correção de defeitos. Portanto, este processo se adapta a qualquer abordagem de depuração (Araki et al., 1991). Embora sua generalidade seja importante, o processo falha em não identificar onde possíveis técnicas e ferramentas podem contribuir para aumentar a eficiência do processo de depuração.

Por outro lado, os demais processos são específicos a alguns problemas encontrados na depuração e as técnicas em particular. Por exemplo, o processo sistemático é dirigido para a solução do problema de redução do espaço de busca através do uso de *fatiamento* dinâmico de programas e de avaliação de hipóteses utilizando execução em reverso. Pan acrescenta a este processo a importância do uso de informação de teste.

O problema de se vincular um processo de depuração a técnicas específicas é que ele é válido somente nas situações em que as técnicas podem ser usadas. No caso de execução em reverso e *fatiamento* dinâmico de programas, há limitações para o uso dessas técnicas em programas reais (aspectos de escalabilidade são discutidos em Resultados e Discussão). O processo de Chan possui o mesmo problema. Embora ele aborde uma questão importante, que é a redução da condição para ocorrência da falha, o processo é projetado para utilizar as técnicas de *fatiamento* de dados e dinâmico de programas.

Portanto, os processos específicos acabam por ter o mesmo problema do processo Hipótese-Validação, que é não indicar onde as técnicas e ferramentas podem contribuir para a melhoria do processo de depuração. A seguir, em Resultados e Discussão, introduz-se processo de *Depuração depois do Teste* cujo objetivo é indicar onde as técnicas e as ferramentas podem contribuir para a melhoria do processo de depuração que ocorre *depois* do teste.

Resultados e Discussão

A característica principal do processo de *Depuração depois do Teste* (DDT) é ser um processo intermediário – não tão genérico como o Hipótese-Validação, nem específico como os anteriormente propostos na literatura – com ênfase no uso de informação de teste em todas as tarefas de localização de defeitos. O processo DDT é descrito no algoritmo apresentado na Fig. 5.

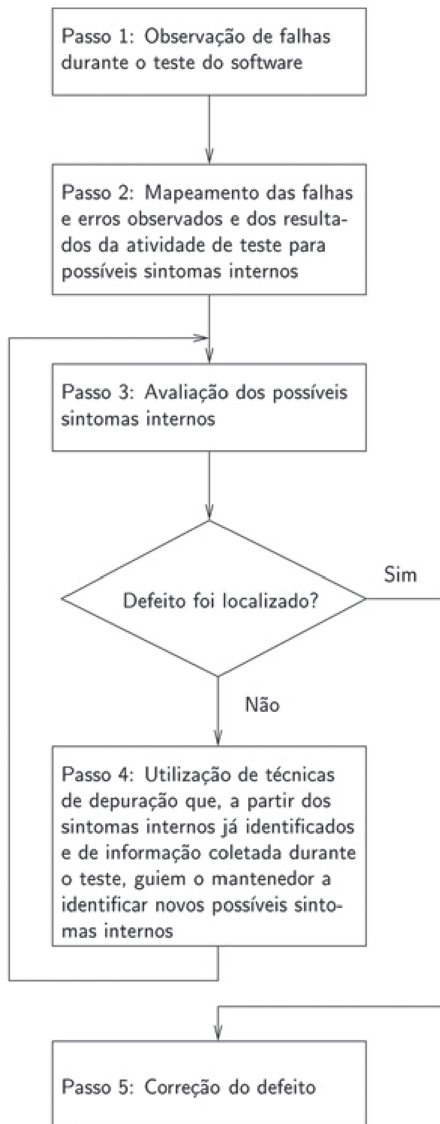

Fig. 5. Processo de Depuração Depois do Teste.

Embora o processo inclua a observação de falhas e a correção do programa, a sua ênfase está na atividade de localização do defeito; por sinal, a mais difícil e custosa (Myers, 1979). O processo indica no passo 2 que são necessárias técnicas que mapeiem as falhas e erros observados e os resultados coletados durante o teste para possíveis sintomas internos. Os possíveis sintomas internos devem então ser avaliados e confirmados como sintomas internos propriamente ditos. No passo 4, é preconizado o uso de técnicas de depuração que ajudem o mantenedor a *refinar* os sintomas internos inicialmente identificados e que acabem por levá-lo a localizar o defeito.

O processo enfatiza o uso de informação de teste durante a depuração. Porém, é necessário qualificar o tipo de informação utilizada. Se ela é obtida por meio de testes não-sistemáticos, então é considerada *básica*. Por exemplo, um relatório de teste que indique tão somente se os casos de teste revelam a presença de defeitos ou não é uma informação *básica*. A informação de teste é *detalhada* quando inclui os produtos gerados durante o teste sistemático do software. Exemplos desses produtos são as entradas e saídas de casos de teste desenvolvidos utilizando técnicas funcionais e os resultados obtidos utilizando teste estrutural.

A seguir é analisada a adequação de várias técnicas de depuração com respeito ao processo DDT. As técnicas são apresentadas resumidamente e avaliadas com relação aos aspectos que o processo mais enfatiza: identificação de possíveis sintomas internos; avaliação dos sintomas; apoio à seleção de novos possíveis sintomas internos; e tipo de informação de teste utilizada. As técnicas revisadas são também analisadas quanto à escalabilidade para sistemas reais.

A depuração baseada em *Rastreamento e Inspeção* é a mais usada na prática. O sucesso dessa técnica deve-se a três fatores: seu uso requer apenas treinamento básico; o custo em tempo de depuração é razoável; e a sua disponibilidade é ampla (presente em qualquer ambiente de programação). Este tipo de depuração envolve o rastreamento de *eventos* e a *inspeção* do estado do programa (no momento em que ocorre um evento).

Na sua forma mais básica, a depuração baseada em rastreamento e inspeção é realizada com a ajuda de comandos de escrita. O mantenedor coloca em pontos estratégicos do programa comandos para imprimir os valores de determinadas variáveis. O objetivo é rastrear um determinado ponto do programa durante a execução (evento) e inspecionar o valor

das variáveis escolhidas (estado parcial do programa). Com os depuradores simbólicos (Stallman & Pesch, 1999; Adams & Muchnick, 1986), o rastreamento de alguns tipos de eventos e a inspeção do estado do programa ficaram extremamente facilitados. Por exemplo, usando os comandos de um depurador simbólico, o mantenedor pode parar a execução do programa no ponto desejado (*breakpoint*), acessar os valores das variáveis, verificar a seqüência de chamadas de procedimentos (*call stack*), atribuir novos valores para variáveis, etc.

Ferramentas mais modernas têm desenvolvido mecanismos de rastreamento de novos eventos e de inspeção de diferentes informações relativas ao estado do programa. DUEL (Golan & Hanson, 1993) é um depurador simbólico que possibilita a especificação de expressões que avaliam o estado do programa. Por exemplo, DUEL permite que o mantenedor consulte quais elementos de um vetor possuem um valor maior, menor ou igual a uma constante ou ao de outra variável. Lancevicius et al. (1997) utilizam uma linguagem de consulta (semelhante às utilizadas em banco de dados) para inspecionar as relações entre as instâncias das classes em uma linguagem orientada a objetos. A ferramenta Coca (Ducassé, 1999), por sua vez, permite o acesso a eventos relacionados às construções da linguagem de programação (e.g., procedimentos, comandos de controle de fluxo) usando uma linguagem declarativa do estilo Prolog. Utilizando a linguagem de consulta de Coca, o mantenedor pode parar a execução do programa em eventos básicos associados à *entrada* ou à *saída* dos comandos de controle de fluxo da linguagem C restringindo, porém, os eventos por meio dos valores das variáveis, do número de ocorrências, etc. Além disso, Coca inclui um mecanismo simplificado de consulta ao estado do programa baseado na linguagem Prolog.

Uma característica interessante de ser incluída nos depuradores simbólicos é a execução em reverso, isto é, permitir que o mantenedor execute o programa em sentido contrário. Há uma longa série de iniciativas de inclusão dessa característica em ferramentas experimentais (Agrawal et al., 1991; Lieberman & Fry, 1998; Balzer, 1969; Tolmach & Appel, 1995); entretanto, não há depuradores simbólicos disponíveis comercialmente com esta função. O fator limitante é que, mesmo nas implementações mais eficientes, é ainda exigida uma grande quantidade de memória para registrar as alterações tanto no estado do programa como nos arquivos de entrada e saída (*I/O logging*). Estes dois requisitos

dificultam a sua utilização em programas reais. Recentemente, algoritmos eficientes para execução bidirecional para depuração de programas foram propostos por Boothe (2000); porém, eles ainda precisam ser avaliados quanto a sua escalabilidade.

Nem todas as ferramentas modernas de rastreamento de eventos e inspeção do estado do programa operam de forma interativa como os depuradores simbólicos. Algumas ferramentas fornecem um relatório depois de terminada a execução do programa (análise *post mortem*) no qual são relatados os eventos observados juntamente com os respectivos estados do programa (e.g., valores de variáveis, endereços de memória). Por exemplo, as ferramentas que permitem verificar o comportamento da memória durante a execução fornecem um relatório indicando os pontos do programa onde ocorreram eventos relacionados com os *erros de memória* (vazamento e acesso inválido) (Hastings & Joyce, 1992; Austin et al., 1994). As ferramentas FORMAN (Auguston, 1995) e CCI (Templer & Jeffery, 1998), por sua vez, permitem que o mantenedor defina eventos a serem monitorados utilizando um conjunto de eventos básicos. Os eventos monitorados são compostos por eventos básicos associados a operações (e.g., atribuição, operações aritméticas, etc.), ao acesso a posições de memória de variáveis em particular, a pontos do programa, etc. O programa é então compilado e instrumentado para monitorar os eventos definidos pelo mantenedor. Depois da execução, um relatório é produzido.

Com relação à escalabilidade para programas reais, os depuradores simbólicos, com exceção daqueles que incluem execução em reverso, são ferramentas utilizadas na depuração de programas dos mais diversos tipos, apesar do impacto no tempo de execução causado pela instrumentação introduzida nos programas. Analogamente, as ferramentas construídas a partir deles (e.g., DUEL, Coca) são igualmente escaláveis. Ferramentas de análise *post-mortem* que realizam rastreamento de erros de memória estão disponíveis comercialmente (Hastings & Joyce, 1992) e são importantes para a depuração de sistemas reais, em especial, na *depuração durante a codificação*. Todavia, outras ferramentas do mesmo tipo (Auguston, 1995; Templer & Jeffery, 1998) podem gerar arquivos de dados (caso os eventos monitorados sejam muito freqüentes) ou programas (caso muitos pontos do programa precisem ser instrumentados) excessivamente grandes.

As ferramentas para rastreamento e inspeção são fundamentais em qualquer abordagem de depuração pois permitem a *avaliação* dos sintomas internos (passo 3 do processo DDT). Além disso, elas também permitem o mapeamento de falhas e erros para sintomas internos (passo 2), especialmente as ferramentas de análise *post mortem*. Entretanto, essas ferramentas utilizam informação básica de teste para realizar as duas tarefas. Com relação à seleção de novos sintomas internos (passo 4), nenhum apoio é fornecido. O mantenedor depende essencialmente da sua experiência para realizar esta tarefa.

A *Depuração com Asserções* é baseada na inclusão de parte da especificação do programa no seu código fonte, de maneira que uma ação é ativada toda vez que a especificação parcial do programa é violada durante a execução. Segundo Staa (2001, 2000), as asserções executáveis reduzem o esforço de localização de defeitos visto que o defeito e seus efeitos (os erros) estão *próximos*, tanto em termos de espaço (distância em número de comandos executados do ponto do programa onde se localiza o defeito até o ponto em que a instrumentação relata o erro), como de tempo (instante entre o alcance do defeito e a observação de seus efeitos).

Esta técnica de depuração foi desenvolvida no final dos anos 60 e algumas linguagens de programação já incluem construções que permitem o seu uso (e.g., a macro pré-definida **assert** do padrão ANSI da linguagem C). Mais recentemente, algumas ferramentas como ASAP (Curcio, 1998) e APP (Rosenblum, 1995) aumentaram ainda mais o poder expressivo das asserções. A seguir, é descrito um exemplo utilizando asserções e a ferramenta APP.

Na Fig.6, o programa anotado com asserções realiza a troca de valores de duas variáveis inteiras sem a utilização de uma variável intermediária utilizando a operação de *ou-exclusivo*. As asserções estão incluídas nos indicadores de comentários especiais `/*@ ... @*/`. A cláusula **assume** define uma pré-condição (válida antes da execução do procedimento); a cláusula **promise** especifica uma pós-condição (válida depois de executada o procedimento); e a cláusula **assert** indica uma condição que deve ser verdadeira no corpo. O operador **in** retorna o valor de uma variável antes da execução do procedimento. A Fig. 7 contém a rotina que é ativada quando a cláusula **promise** é violada.

A depuração utilizando asserções permite o mapeamento de erros para sintomas internos pois elas indicam pontos do programa onde ocorrem

discrepâncias em relação à especificação. Portanto, esta técnica de depuração apóia o passo 2 do processo DDT; porém, utiliza informação básica de teste. Segundo Rosenblum (1995), o custo em termos de espaço e tempo de execução dos programas anotados com asserções é insignificante, o que viabiliza a sua utilização em sistemas reais. Entretanto, esta técnica requer a codificação adicional de parte da especificação que, por sua vez, poderá também conter defeitos.

```
void swap(x,y)
{
int * x;
int * y;
/*@
    assume x && y && x != y;
    promise *x == in *y;
    promise *y == in *x;
@*/
{
    *x = *x ^ *y;
    *y = *x ^ *y;
/*@
    assert *y == in *x;
@*/
}
```

Fig. 6. Uso de asserções (Rosenblum, 1995).

```
promise * x == in * y { printf("%s invalid:file %s,
",__ANNONAME__,__FILE__); printf("line %d, function
%s:\n",__ANNOLINE__,__FUNCTION__); printf("out *x
== %d, out *y == %d\n",*x,*y); }
```

Fig. 7. Rotina ativada pela violação de uma asserção (Rosenblum, 1995).

A atividade de depuração pode ser facilitada se o mantenedor puder dirigir a sua atenção para um trecho relativamente pequeno de código onde o defeito está localizado. O *fatiamento* de programas tem como objetivo reduzir o espaço de busca do defeito por meio da análise estática ou dinâmica do programa. Essa técnica foi inicialmente proposta do ponto de vista estático por Weiser (1984) e, posteriormente, do ponto de vista dinâmico por Agrawal & Horgan (1990) e Korel & Laski (1988). Essas idéias originais foram espandidas de várias maneiras e para diferentes contextos. Tip (1995) e Kamkar (1995) possuem excelentes revisões bibliográficas que discutem vários aspectos da técnica (e.g., *fatiamento* estático e dinâmico, algoritmos, estrutura de dados, *fatiamento* intra e inter-procedimental, tratamento de ponteiros e de programas não-estruturados, custo, etc.).

Durante a elaboração desse texto, optamos por traduzir os termos originais *slicing* (*fatiamento*) e *slice* (*fatia*) com o objetivo de colaborar para o estabelecimento de um jargão de depuração de software em língua portuguesa. A seguir é discutida a aplicação do *fatiamento* de programas na depuração.

O *fatiamento* (*slicing*) de programas é uma técnica cujo objetivo é identificar fatias (*slices*) do programa. Uma *fatia* é um conjunto de comandos que afetam os valores de uma ou mais variáveis em um determinado ponto do programa. As variáveis e o ponto do programa definem o *critério de fatiamento*. A *fatia* do programa pode ser determinada *estaticamente* ou *dinamicamente*. No primeiro caso, os comandos selecionados *podem* afetar as variáveis no ponto especificado para alguma possível entrada do programa; no segundo caso, os comandos selecionados *efetivamente* afetam os valores das variáveis no ponto especificado para uma determinada entrada.

Tanto as *fatias* estáticas como as dinâmicas podem ser executáveis ou não. Se executável, a *fatia* é um subconjunto dos comandos do programa original que também é um programa. Este novo programa fornece os mesmos valores para as variáveis selecionadas no ponto especificado do programa original. As *fatias* executáveis em geral são *maiores* porque precisam incluir declarações de variáveis e outros comandos que não afetam o critério de *fatiamento*, mas que são necessários para execução.

A técnica de *fatiamento* de programas apóia dois aspectos do processo DDT: mapeamento de falhas para possíveis sintomas internos (passo 2) e a seleção de novos possíveis sintomas a partir daqueles inicialmente

identificados (passo 4); porém, nas duas tarefas é utilizada informação básica de teste. Do ponto de vista prático, porém, esta técnica possui alguns problemas que têm impedido a sua utilização em situações reais. O primeiro é o *tamanho* das *fatias*, tanto estáticas (especialmente) como dinâmicas. Para programas grandes, o número de comandos que *afetam* um critério de *fatiamento* pode também ser grande, o que torna a técnica pouco atrativa. Para superar este problema em parte, Korel & Rilling (1997) desenvolveram, internamente aos procedimentos, *fatias* dinâmicas parciais (e.g., *fatia* do código de um laço) de forma a restringir o tamanho do *pedaço* de código que o mantenedor terá de investigar; e, em termos de sistema, os autores desenvolveram *fatias* de informação relativa à interação dos procedimentos (e.g., *fatia* do grafo de chamada) (Korel & Rilling, 1998). O segundo, e mais importante, problema da técnica é o seu custo. O *fatiamento* dinâmico de programas requer o monitoramento das posições de memória de maneira a identificar precisamente os comandos que afetam um critério de *fatiamento*. Este requisito, porém, impõe um custo muito grande em tempo de depuração para programas que possuem longas execuções (Nishimatsu et al., 1999; Wong et al., 1999).

Pan (DeMillo et al., 1996; Pan, 1993) propõe a identificação de um novo tipo de *fatia*, chamada *fatia crítica*, durante o teste baseado em defeitos (análise de mutantes). Este novo tipo de *fatia* é definido da seguinte maneira. Suponha-se que um programa *P* tenha produzido um valor incorreto para uma variável de saída *v*. Seja *M* uma versão alterada de *P* (mutante) em que apenas um comando *S* tenha sido eliminado. Se o valor de *v* é diferente quando *M* é executado com um caso de teste então ele é um *comando crítico*. A *fatia* crítica é composta pelos *comandos críticos* do programa *P*.

O custo de determinação da *fatia* crítica isoladamente é muito alto visto que, para um programa com *n* comandos, seria necessário executar *n* programas mutantes *M* com cada caso de teste (DeMillo et al., 1996). Entretanto, este custo pode ser amortizado durante o teste se a *fatia* crítica for obtida durante o teste com o critério análise de mutantes utilizando o operador *eliminação de comando*. DeMillo et al. (1996) descrevem um experimento com programas pequenos em que as *fatias* críticas selecionaram 25% menos comandos que as *fatias* dinâmicas.

Agrawal et al. (1998); Agrawal et al. (1995) propõem a determinação de uma *fatia* do programa a partir dos resultados obtidos do teste estrutural do programa. O conjunto de comandos associados aos requisitos de teste

estruturais (e.g., *nós*, *ramos* e *adu*s) executados por um caso de teste particular é chamado de *fatia* de execução.

A vantagem dos *fatiamentos* baseados em informação de teste é que a maior parte do custo para obtenção das *fatias* já foi amortizado durante o teste do programa. Entretanto, semelhantemente aos outros *fatiamentos* de programas, há uma grande probabilidade das *fatias* derivadas de informação de teste incluírem um número elevado de comandos. As *fatias* críticas, apesar da possível redução de 25% trazida em relação às *fatias* dinâmicas, muito provavelmente incluem um grande trecho de código quando determinadas para programas complexos e críticos (os mais indicados para o teste com o critério análise de mutantes). Já as *fatias* de execução são sempre iguais ou maiores que as *fatias* dinâmicas.

Tanto o *fatiamento* crítico quanto o de execução são úteis para mapear informação detalhada de teste para possíveis sintomas internos (comandos suspeitos); portanto, essas duas técnicas apóiam o passo 2 do processo DDT. Porém, elas não apóiam a seleção de novos sintomas internos (passo 4) devido à maneira como as *fatias* são determinadas. No caso das *fatias* críticas, elas podem ser determinadas apenas para as variáveis de saída verificadas durante o teste de mutantes; as *fatias* de execução, por sua vez, são determinadas em relação a um caso de teste e não a um critério de *fatiamento*.

Heurísticas utilizando *fatias* têm sido propostas para reduzir o espaço de busca para localização do defeito (Collofello & Cousins, 1987; Pan & Spafford, 1992; Lyle & Weiser, 1987; Agrawal, 1991; Agrawal et al., 1995; Chen & Cheung, 1997; Pan, 1993). A idéia é realizar operações com as *fatias* para determinar um conjunto menor de comandos com grande probabilidade de conter o defeito. As heurísticas mais simples realizam operações de intersecção e união de *fatias* (Pan & Spafford, 1992; Pan, 1993).

Lyle & Weiser (1987) introduziram o *recorte* de *fatias* estáticas. Os autores propõem a *subtração* das *fatias* obtidas de variáveis de saída *corretas* (variáveis cujos valores estão corretos para todos os casos de teste) das *fatias* de variáveis de saída *incorretas* (em pelo menos um caso de teste produziram um valor incorreto). A intuição subjacente é que a eliminação dos comandos comuns a ambas as *fatias* pode levar à identificação de um conjunto menor de comandos com maior chance de conter o defeito.

De maneira análoga, as *fatias* dinâmicas podem ser utilizadas para a obtenção de *fragmentos de recorte* (Pan & Spafford, 1992; Agrawal, 1991; Chen & Cheung, 1997; Pan, 1993). Neste caso, os *fragmentos de recorte* podem ser calculados usando *fatias* das mesmas variáveis, não necessariamente de saída, que tenham produzido valores corretos e incorretos em dois casos de teste diferentes. A técnica de *recorte* pode ainda envolver a união ou intersecção de *fatias* de variáveis obtidos de casos de teste reveladores de defeito menos a união ou intersecção de *fatias* de casos de teste não-reveladores de defeito. Os termos *recorte* e *fragmento de recorte* foram traduzidos a partir dos termos originais *dicing* (*fatiamento em cubos*) e *dice* (*cubo*, *dado*), respectivamente.

Outra maneira de identificar heuristicamente comandos do programa com grande probabilidade de conter o defeito é pelo estabelecimento de um *ranking* entre eles. Neste *ranking*, os comandos que ocorrem mais freqüentemente em *fatias* de casos de teste que manifestam falhas são mais bem classificados do que aqueles que não ocorrem tão freqüentemente (Pan & Spafford, 1992; Pan, 1993). Esta idéia foi inicialmente proposta por Collofello & Cousins (1987) para evitar que um defeito localizado em um trecho do código executado tanto por casos de teste reveladores de defeito como não-reveladores de defeito fosse eliminado na operação de subtração da técnica de *recorte*.

Estes autores definem dez heurísticas que realizam operações de *recorte* e de *ranking* de nós (obtidos durante o teste com o critério todos nós) para identificar trechos de código suspeitos (Collofello & Cousins, 1987). Agrawal et al. (1995) revisitaram esta abordagem para a definição das *fatias* de execução e também de heurísticas baseadas em *recorte* de outros requisitos de teste (e.g., *ramos*, *adu*s) executados pelos casos de teste.

O uso de heurísticas impõe alguns riscos. Apesar de a intuição ser de que há grande probabilidade do trecho de código selecionado conter o defeito, há também a chance dele ser excluído durante as operações envolvendo conjuntos (e.g., intersecção, subtração) ou durante a criação do *ranking*. Neste último caso, o trecho selecionado inclui os *efeitos* do defeito, mas exclui o próprio. Além disso, as heurísticas que operam sobre *fatias* possuem as mesmas restrições inerentes à técnica utilizada para obtê-las.

Do ponto de vista do processo DDT, as heurísticas que utilizam *fatias* estáticas e dinâmicas, da mesma maneira que a técnica de *fatiamento* de

programas, apóiam as tarefas definidas nos passos 2 e 4 utilizando informação básica de teste. Já as heurísticas que utilizam informação de teste apóiam somente a tarefa de mapeamento para sintomas internos (passo 2), utilizando, porém, informação detalhada de teste.

A técnica de *depuração algorítmica* foi originalmente desenvolvida para programas sem efeitos colaterais escritos em Prolog (Shapiro, 1983). Esta técnica é baseada em um processo interativo durante o qual o mantenedor fornece *conhecimento* a respeito do comportamento esperado do programa ao sistema de depuração; e este, por sua vez, guia o mantenedor durante o processo de localização do defeito (Fritzson et al., 1992).

A técnica funciona da seguinte maneira. Para localização do defeito, o caso de teste que manifestou uma falha deve ser executado sob a supervisão do sistema baseado em depuração algorítmica. O sistema cria então uma árvore de execução na qual os nós representam invocações dos procedimentos do programa. Esses nós contêm o nome do procedimento e os valores de entrada e saída utilizados na invocação em particular. O sistema então visita, partindo do procedimento de mais alto nível, os nós da árvore de execução perguntando ao mantenedor se os valores dos parâmetros de entrada e saída estão corretos. Se sim, o processo continua visitando os próximos nós de *mesmo* nível; caso contrário, o processo visita os nós dos procedimentos de nível *inferior* invocados pelo procedimento de nível superior. Este processo termina quando é identificado um procedimento no qual os parâmetros de entrada estão *corretos* e os de saída *incorretos* ou que não faz chamada a nenhum outro procedimento ou, se o faz, os valores dos parâmetros de entrada e saída dos procedimentos invocados estão corretos.

Caso o mantenedor responda corretamente às questões colocadas pelo sistema de depuração algorítmica, o procedimento onde está o defeito é identificado. Entretanto, a aplicabilidade dessa técnica em programas reais é reduzida. Entre as dificuldades para sua utilização estão: o número de perguntas que o mantenedor deve responder; o tratamento de efeitos colaterais, especialmente os causados pelo uso de ponteiros; e o espaço (não-limitado) requerido pela árvore de execução cujo tamanho depende do número de invocações dos procedimentos.

Fritzson et al. (1992) desenvolveram um sistema de depuração algorítmica para a linguagem Pascal (*GADT — Generalized Algorithmic Debugging and Testing*) no qual a técnica de *fatiamento* dinâmico de programas e informação de teste funcional (Ostrand & Balcer, 1988) são usados para

reduzir o número de perguntas a serem respondidas pelo mantenedor. GADT funciona da seguinte maneira. Suponha-se que um procedimento *P* tenha produzido um valor incorreto para o parâmetro de saída *q*. GADT determina usando *fatiamento* dinâmico (Kamkar, 1998) as invocações dos procedimentos chamados por *P* que influenciaram o valor *q*, de forma que somente estas invocações são visitadas durante o processo de depuração. Além disso, antes de perguntar se os parâmetros de entrada e saída de uma invocação de procedimento estão corretos, GADT verifica em uma base de dados se os valores dos parâmetros de entrada e saída já foram executados por algum caso de teste não-revelador de defeito ou se pertencem a uma *categoria* que contém apenas esse tipo de caso de teste; em caso afirmativo, GADT também ignora esse procedimento e visita o próximo.

O sistema GADT procura reduzir um dos problemas da técnica de depuração algorítmica, que é o número de interações com o mantenedor. Entretanto, outros problemas importantes como o tamanho da árvore de execução e o tratamento de efeito colaterais causados por ponteiros não são tratados, o que ainda torna sua aplicabilidade reduzida (Lian et al., 1997).

Com relação ao processo DDT, a técnica de depuração algorítmica apóia essencialmente a identificação de novos possíveis sintomas internos até a localização do procedimento *defeituoso* (passo 4). A definição da técnica não prevê o uso de informação detalhada de teste; porém, ela pode ser útil à depuração algorítmica, como evidenciado pelo sistema GADT.

A técnica de depuração algorítmica como proposta por Shapiro (1983) visa à idenficação do procedimento onde se encontra o defeito; entretanto, ela não apóia a localização *internamente* ao procedimento. Korel (1988) utiliza as técnicas de depuração algorítmica e *fatiamento* de programas para estabelecer uma estratégia para encontrar o defeito dentro do procedimento. A ferramenta PELAS (*Program Error-Locating Assistant System*) implementa esta estratégia. Outras ferramentas como Spyder (Agrawal, 1991; Viravan, 1994) e FIND (Shimomura et al., 1995; Shimomura, 1993) adotam abordagens semelhantes variando o mecanismo de interação com o usuário e a técnica de *fatiamento* utilizada. A seguir, é mostrado como a ferramenta PELAS é utilizada para descrever a localização de defeitos baseada em depuração algorítmica e *fatiamento* de programas.

PELAS analisa estaticamente um procedimento identificado como defeituoso e produz durante a execução do programa um grafo chamado

rede de dependência. Os nós da rede de dependência representam os pontos de execução do caso de teste; e os ramos representam as relações entre os nós. Essas relações são descritas a seguir:

1. **Influência de Dados**: relação estabelecida entre a definição de uma variável *v* no ponto de execução X^p e o subseqüente uso de *v* no ponto Y^q, desde que não haja nenhuma definição de *v* nos comandos executados entre X^p e Y^q.
2. **Influência de Controle**: é a relação entre o ponto de execução X^p, onde *X* é um comando de controle de fluxo, e os demais pontos cuja execução é determinada pelo resultado da avaliação do predicado do comando de controle de fluxo *X* no ponto X^p.
3. **Influência Potencial**: considere-se o ponto de execução X^p, onde *X* é um comando de controle de fluxo, e o ponto de execução Y^q tal que existe um uso de *v* em Y^q e não há nenhuma definição de *v* nos comandos executados entre X^p e Y^q. Se existe um caminho alternativo entre *X* e *Y* tal que, se este caminho fosse executado, ocorreria uma redefinição de *v*, então entre X^p e Y^q há uma relação de *influência potencial*.

Considere-se o exemplo contido na Fig. 8 para ilustrar o algoritmo de localização implementado em PELAS. As entradas `n = 2` e `a = (6,2)` produzem a saída incorreta `s` = 4 (a saída correta é `s` = 8). A Fig. 9 contém os pontos de execução do programa para este caso de teste; e a Fig. 10 contém a rede de dependência.

N.º	Programa **prog_sum**
1	`get(n,a);`
2	`t:=1;   { correto: t:=10 }`
3	`s:=a [ 1] ;`
4	`i:=2;`
5	**while** `i <= n` **loop**
6	`s:=s-a [ i] ; { correto: s:=s+a [ i] ; }`
7	`i:=i+1;`
	end loop;
8	**if** `s > t` **then**
9	**if** `s mod 2 != 0` **then**
10	`s:=s+1;`
	end if;
	end if;
11	`put(s); <DIV ALIGN="CENTER">`

Fig. 8. Programa desenvolvido por Shimomura (1993).

Comando X^p	Código Fonte	Valores
1^1	`get(n,a);`	{n=2; a=(6,2)}
2^2	`t:=1;`	{t=1}
3^3	`s:=a [ 1] ;`	{s=6}
4^4	`i:=2;`	{i=2}
5^5	**`while`** `i <= n` **`loop`**	{2 <= 2}
6^6	`s:=s-a [ i] ;`	{s=2-a[2]=6-2=4}
7^7	`i:=i+1;`	{i=3}
5^8	**`while`** `i <= n` **`loop`**	{3 <= 2}
8^9	**`if`** `s > t` **`then`**	{ 4 > 1}
9^{10}	**`if`** `s mod 2 != 0` **`then`**	{ 4 mod 2 !=0}
11^{11}	`put(s);`	{ put(4) }

Fig. 9. Execução do programa da Fig. 8 para o caso de teste `n` = 2 e `a` = (6,2).

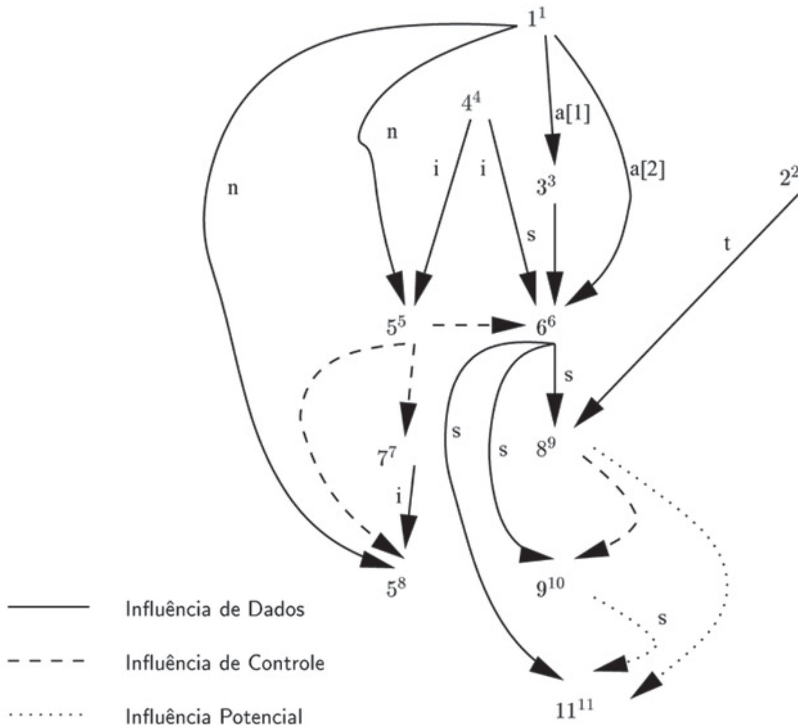

Fig. 10. Rede de dependência gerada por PELAS durante a execução do programa.

O sintoma interno relacionado com a falha observada é o valor da variável `s` no ponto de execução 11^{11}. A partir da rede de dependência, PELAS indica ao mantenedor três pontos de execução – 6^{6}, 8^{9} e 9^{10} – que influenciam o resultado de `s` em 11^{11}. O mantenedor deve então escolher um dos três para continuar a depuração. Suponha-se que ele escolha o ponto 6^{6}. PELAS recupera o estado do programa neste ponto e o mantenedor é questionado se o valor de `s` igual a 4 em 6^{6} é correto. Neste caso ele está incorreto, então o sistema pergunta se os valores de entrada (`s` = 2, `a [ 2]` =2) em 6^{6} estão corretos. A resposta é sim, estão corretos; logo, o defeito foi localizado no comando de número 6 do programa.

A estratégia de depuração implementada por PELAS, Spyder e FIND depende essencialmente da construção de um grafo em tempo de depuração que registra *cada* ponto de execução, bem como as suas relações. A complexidade espacial para construção do grafo é não-limitada, visto que depende do número de vezes que os comandos do procedimento são executados. Esta questão é importante para programas com longas execuções, pois a memória disponível pode ser totalmente consumida (Korel & Yalamanchili, 1994).

Os grafos construídos por PELAS, Spyder e FIND tinham inicialmente o objetivo de determinar *fatias* dinâmicas do programa. Novos algoritmos para determinação de *fatias* dinâmicas que não dependem da construção do grafo foram desenvolvidos (Gyimóthy et al., 1999; Korel & Yalamanchili, 1994). Porém, essas soluções, por evitarem a construção do grafo, perdem as instâncias dos comandos que influenciam um determinado ponto da execução, o que impede a seleção de novos sintomas internos (pontos de execução) a serem investigados. Some-se a isto o fato de o custo em tempo de depuração para determinação de *fatias* dinâmicos (usando grafos ou não) ser ainda proibitivo para programas com longas execuções (Wong et al., 1999; Nishimatsu et al., 1999).

A estratégia de depuração implementada em PELAS, Spyder e FIND expandem a idéia de depuração algorítmica (originalmente utilizada para a determinação do procedimento defeituoso) para localização de defeitos internamente aos procedimentos. Portanto, ela apóia o passo 4 do processo DDT; porém, utilizando informação básica de teste.

A Tabela 1 apresenta os passos do processo DDT que cada técnica de depuração apóia bem como o tipo de informação de teste que utiliza e a sua escalabilidade para situações reais. Uma observação imediata da tabela é que as várias técnicas de depuração devem ser combinadas para

apoiar completamente o processo DDT. Isto é esperado visto que as técnicas dão ênfase a diferentes problemas colocados pelos passos do processo.

Tabela 1. Adequação das técnicas de depuração ao processo DDT.

Técnica de Depuração	*Passo do Processo DDT*			*Informação de Teste*	*Escalabilidade Sistemas Reais*
	2	*3*	*4*		
Rastreamento e Inspeção	√	√		básica	√
Asserções	√			básica	√
Fatiamento Estático	√		√	básica	
Fatiamento Dinâmico	√		√	básica	
Fatiamento Informação de Teste	√			detalhada	√
Heurísticas de *Fatias* Estático	√		√	básica	
Heurísticas de *Fatias* Dinâmico	√		√	básica	
Heurísticas de *Fatias* de Informação de Teste	√			detalhada	√
Depuração Algorítmica Inter-Procedimental	√		√	detalhada	
Depuração Algorítmica Intra-Procedimental	√		√	básica	

A técnica de depuração baseada em rastreamento e inspeção é fundamental ao processo DDT, pois a *avaliação dos possíveis sintomas internos* (passo 3) é apoiada apenas pelas ferramentas que apóiam esta técnica. Entretanto, essas ferramentas utilizam informação básica de teste. Nesse sentido, elas não são completamente adequadas à depuração que ocorre *depois* do teste, visto que não tiram proveito da informação coletada durante essa atividade. No contexto do processo DDT, as ferramentas de rastreamento e inspeção devem ser capazes de rastrear eventos relacionados com os resultados obtidos do teste sistemático, de forma a permitir a análise dos possíveis sintomas internos indicados por eles. Por exemplo, os requisitos de teste de critérios estruturais de teste (ramos, *adu*s e *adpu*s) que são exercitados durante a execução de um caso de teste podem ser entendidos como compostos de *eventos* de teste nos quais o estado do programa pode ser inspecionado. Possíveis eventos de teste associados a uma *adu* seriam o alcance dos pontos de execução onde ocorrem a definição e o uso da variável. No entanto, as atuais ferramentas de rastreamento e inspeção não possuem mecanismos para rastreamento desses eventos.

As técnicas de *fatiamento* estático e dinâmico de programas apóiam o *mapeamento de falhas e erros para possíveis sintomas internos* (passo 2) e a *seleção de novos possíveis sintomas* (passo 4). Porém, essas técnicas possuem alguns problemas: não são escaláveis e não utilizam informação de teste detalhada. Um dos problemas de escalabilidade é o número muito grande de comandos suspeitos contidos nas *fatias* selecionadas. O uso de diferentes tipos de *fatias* (Korel & Rilling, 1997; Korel & Rilling, 1998) e heurísticas (Pan & Spafford, 1992; Chen & Cheung, 1997) pode minorar esse problema. Todavia, o segundo problema, relacionado com o custo de obtenção das *fatias*, continua sendo um fator impeditivo para o uso das técnicas em situações reais. O fato de não utilizar informação de teste, por sua vez, torna as *fatias* estáticas e dinâmicas menos adequadas ao processo DDT e também implica que todo o custo para obtê-las acontece somente durante a depuração, não podendo ser amortizado em outras fases do processo de desenvolvimento de software.

As *fatias* de programas obtidas de informação de teste são as mais adequadas ao processo DDT visto que são determinadas utilizando resultados coletados durante o teste e o processo preconiza o uso desse tipo de informação. A vantagem é que elas podem ser obtidas quase que diretamente durante a depuração, pois o custo de obtenção dos resultados de teste já foi amortizado e os algoritmos para determinação das *fatias* são baratos (basicamente realizam o mapeamento da informação de teste para um trecho de código). Nesse sentido, os *fatiamentos* baseados em informação de teste são os que possuem maiores perspectivas de escalabilidade para programas reais (Wong et al., 1999). Entretanto, eles não apóiam *a seleção de novos sintomas internos* (passo 4). Este problema é importante visto que o trecho de código associado as *fatias* baseadas em informação de teste pode ser grande. A utilização de heurísticas diminui o tamanho do trecho de código a ser examinado, porém, há sempre a possibilidade de o mantenedor ter a sua atenção direcionada para um ponto do programa que não contém o defeito ou que indica os *efeitos* do defeito, e não o próprio. Portanto, mecanismos para refinamento da informação de teste, visando a sua utilização na identificação de novos possíveis sintomas internos, são necessários no contexto de *depuração depois do teste*.

A técnicas de depuração algorítmica, como as técnicas de *fatiamento* estático e dinâmico, apóiam os passos 2 e 4. Porém, tanto a depuração

algorítmica inter-procedimental como intra-procedimental têm a sua aplicabilidade restringida pelo grafo de tamanho não-limitado utilizado nessas técnicas, o que as torna inviáveis em um contexto industrial de produção de software.

Uma constatação da análise realizada é que o uso de informação detalhada de teste na depuração resulta em técnicas de baixo custo em tempo de depuração, desde que o teste tenha sido realizado de maneira sistemática. Nesse sentido, elas são as que apresentam maiores possibilidades de utilização em ambientes industriais. No entanto, apenas um passo do processo DDT é apoiado de maneira eficiente por essas técnicas, corroborando a afirmação de Harrold (2000) de que o uso de informação de teste na depuração encontra-se ainda na sua *infância*. É necessário, portanto, o desenvolvimento de estratégias de depuração que utilizem a informação coletada durante o teste em todos os passos do processo DDT.

Conclusões

Neste trabalho, a atividade de depuração que ocorre *depois* do teste de software foi analisada. Inicialmente, vários processos de depuração foram analisados e o processo de *Depuração depois do Teste* (DDT), voltado para a identificação das tarefas de depuração que ocorrem *depois* do teste, foi definido. O processo DDT ressalta os seguintes aspectos (passos): identificação de possíveis sintomas internos; avaliação dos sintomas identificados; apoio à seleção de novos sintomas; e tipo de informação de teste utilizada. As principais técnicas de depuração de programas procedimentais foram avaliadas tendo como guia o processo DDT e a sua escalabilidade para sistemas reais.

Dessa avaliação, observou-se que os resultados do teste sistemático de software, quando utilizados na depuração, resultam em técnicas de baixo custo e com maiores perspectivas de escalabilidade para programas reais. Infelizmente, esses resultados têm sido utilizados apenas para apoiar um dos passos do processo DDT - *identificação de possíveis sintomas internos* (passo 2). Entretanto, para que a *depuração depois do teste* seja completamente apoiada, é importante que os demais passos também utilizem informação detalhada de teste. Assim, identificou-se a necessidade de estratégias de depuração que utilizem, de maneira eficaz e eficiente, as informações de teste na *avaliação de possíveis sintomas internos* (passo 3) e na *seleção de novos possíveis sintomas internos* (passo 4).

Referências Bibliográficas

ADAMS, E.; MUCHNICK, S. S. Dbxtool: a window-based symbolic debugger for sun workstation. **Software Practice and Experience**, v. 16, n. 7, p. 653-669, 1986.

AGRAWAL, H. **Towards automatic debugging of computer programs**. 1991. Thesis (Doctor of Philosophy) - Purdue University, West Lafayette.

AGRAWAL, H.; ALBERI, J. L.; HORGAN, J. R.; LI, J. J.; LONDON, S.; WONG, W. E.; GOSH, S.; WILDE, N. Mining system tests to aid software maintenance. **IEEE Computer**, v. 31, n. 7, p. 64-73, 1998.

AGRAWAL, H.; DEMILLO, R. A.; SPAFFORD, E. H. An execution-backtracking approach to debugging. **IEEE Software**, v. 8, n. 3, p. 21-26, 1991.

AGRAWAL, H.; HORGAN, J. R. Dynamic program slicing. **ACM SIGPLAN Notices**, v.25, n. 6, p. 246-256, June, 1990.

AGRAWAL, H.; HORGAN, J. R.; LONDON, S.; WONG, W. E. Fault localization using execution slices and dataflow tests. In: IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING - ISSRE'95, 6., 1995, Toulouse. **Proceedings**... [S.l.: IEEE, 1995]. p. 143-151.

ARAKI, K.; FURUKAWA, Z.; CHENG, J. A general framework for debugging. **IEEE Software Magazine**, v. 8, n. 3, p. 14-20, 1991.

AUGUSTON, M. A program behavior model based on event grammar and its application for debugging automation. In: INTERNATIONAL WORKSHOP ON AUTOMATED AND ALGORITHMIC DEBUGGING - AADEBUG'95, 2., 1995, Saint-Malo. **Proceedings**... Rennes: IRISA, 1995. p. 277-291.

AUSTIN, T. M.; BREACH, S. E.; SOHI, G. S. Efficient detection of all pointer and array access error. **ACM SIGPLAN Notices**, v. 29, n. 6, p. 290-301, June, 1994.

BALZER, R. M. Exdams: Extensible debugging and monitoring system. In: AFIPS SPRING JOINT COMPUTER CONFERENCE, 1969, Boston. **Proceedings**... Arlington: AFIPS Press, 1969. p 567-580.

BOOTHE, B. Efficient algorithms for bidirectional debugging. **ACM SIGPLAN Notices**, v. 35, n. 5, p. 299-310, May, 2000.

BUDD, T. A. Mutation analysis: ideas, example, problems and prospects. In: CHANDRASEKARAN, B.; RADICCHI, S. (Ed.). **Computer program testing**. Amsterdam: North-Holland, 1981.

CHAIM, M. L.; MALDONADO, J. C.; JINO, M.; NAKAGAWA, E. Y. POKE-TOOL — estado atual de uma ferramenta para teste estrutural de software baseado em análise de fluxo de dados. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 13.; SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 12., 1998, Maringá. **Caderno de ferramentas**. Maringá: SBC, 1998. p. 37-45.

CHAN, T. W. A framework for debugging. **Journal of Computer Information Systems**, v. 38, 1, p. 67-73, 1997.

CHAN, T. W.; LAKHOTIA, A. Debugging program failure exhibited by voluminous data. **Journal of Software Maintenance**: Research and Practice, v. 10, p. 111-150, 1998.

CHEN, T. Y.; CHEUNG, Y. Y. On program dicing. **Software Maintenance**: Research and Practice, v. 9, p. 33-46, 1997.

COLLOFELLO, J. S.; COUSINS, L. Toward automatic software fault localization through decision-to-decision path analysis. In: NATIONAL COMPUTER CONFERENCE, 56., 1987, Chicago. **Proceedings of the AFIP 1987**. [Chicago: s. n.]: 1987. p. 539-544.

CURCIO, I. D. ASAP — a simple assertion pre-processor. **ACM SIGPLAN Notices**, v. 33, n. 12, p. 44-51, 1998.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: help for the practicing programmer. **Computer**, v. 11, n. 4, p. 34-43, 1978.

DEMILLO, R. A.; PAN, H.; SPAFFORD, E. H. Critical slicing for software fault localization. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 1996, San Diego. **Proceedings**... New York: ACM Press, 1996. p. 121-134.

DUCASSÉ, M. Coca: an automated debugger for C. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 21., 1999, Los Angeles. **Proceeding**s... Los Alamitos: IEEE Computer Society Press, 1999. p. 504-513.

FRANKL, F. G.; WEISS, S. N.; WEYUKER, E. J. ASSET—a system to select and evaluate test. In: IEEE CONFERENCE ON SOFTWARE TOOLS, 1985, New York. **Proceedings**... Los Alamitos: IEEE Computer Society Press, 1985. p. 72-79.

FRANKL, F. G.; WEYUKER, E. J. An applicable family of data flow testing criteria. **IEEE Transactions on Software Engineering**, v. 14, n. 10, p. 1483-1495, 1988.

FRITZSON, P.; SHAHMEHRI, N.; KAMKAR, M.; GYIMOTHY, T. Generalized algorithmic debugging and testing. **ACM Letters on Programming Languages and Systems**, v. 1, n. 4, p. 303-322, 1992.

GHEZZI, C.; JAZAYERY, M. **Programming languages concepts**. 2nd ed. New York: John Wiley, 1987.

GOLAN, M.; HANSON, D. DUEL - a very high-level debugging language. In: USENIX ASSOCIATION. **Proceedings of the Winter 1993 Usenix Conference**. Berkeley, 1993. p. 107-117.

GYIMÓTHY, T.; BESZÉDES, A.; FORGÁCS, I. An efficient relevant slicing method for debugging. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE, 7.; ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, 7., 1999, Toulouse. **Proceedings**... [Berlin]: Springer, 1999. p. 303-321. (Lecture Notes in Computer Science).

HARROLD, M. J. Testing: a roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 22., 2000, Limerick. **Proceedings of the conference on the future of software engineering**. New York: ACM Press, 2000. p. 61-72.

HASTINGS, R.; JOYCE, B. Purify: fast detection of memory leaks and access errors. In: : USENIX ASSOCIATION. **Proceedings of the Usenix Winter 1992 technical conference**. San Francisco: Usenix Association, 1992. p. 125-138.

HERMAN, P. M. Data flow approach to program testing. **Australian Computer Journal**, v. 8, n. 3, p. 92-96, 1976.

HORGAN, J. R.; LONDON, S. Data flow coverage and the C language. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING ANALYSIS, 1991, Victoria, Canada. **Proceedings of the symposium on testing, analysis, and verification**. New York: ACM Press, 1991. p. 87-97.

IEEE COMPUTER SOCIETY. **IEEE software engineering standards collection 1991 edition**. New York, 1991.

KAMBAR, M. An overview and comparative classification of program slicing techniques. **Journal of Systems and Software,** v. 31, p. 197-214, 1995.

KAMBAR, M. Application of program slicing in algorithmic debugging. **Information and Software Technology**, v. 40, p. 637-645, 1998.

KOREL, B. PELAS - Program Error-Locating Assistant System. **IEEE Transactions on Software Engineering**, v. 14, n. 9, p. 1253-1260, 1988.

KOREL, B.; LASKI, J. Dynamic program slicing. **Information Processing Letters**, v. 29, n. 3, p. 155-163, 1988.

KOREL, B.; RILLING, J. Application of dynamic slicing in program debugging. In: WORKSHOP ON AUTOMATED AND ALGORITHMIC DEBUGGING, 3., 1997, Linköping. **AADEBUG'97**: proceedings of the third International Workshop on Automatic Debugging. Abstract. Disponível em: <http://www.ep.liu.se/ea/cis/1997/009/05>. Acesso em: set. 2002. Disponível também em: **Linköping Electronic Articles in Computer and Information Science,** v. 2, n. 9-05, p.59-74, 1997.

KOREL, B.; RILLING, J. Program slicing in understanding large programs. In: INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION, 6., 1998, Ischia. **Proceedings**... [New York]: IEEE, [1998]. p. 145-152.

KOREL, B.; YALAMANCHILI, S. Forward computation of dynamic program slices. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 1994, Seatle. **Proceedings**... New York: ACM Press, 1994. p. 66-79.

LANCEVICIUS, R.; HÖLZLE, U.; SINGH, A. Query-based debugging of object-oriented programs. **ACM SIGPLAN Notices**, v. 32, n. 10, p. 304 – 317, Oct. 1997.

LASKI, J.; KOREL, B. A data flow oriented program testing strategy. **IEEE Transactions on Software Engineering**, v. 9, n. 3, p. 347-354, 1983.

LIAN, L.; KUSUMOTO, S.; KIKUNO, T.; MATSUMUTO, K.; TORII, K. A new fault localizing method for the program debugging process. **Information and Software Technology**, v. 39, p. 271-284, 1997.

LIEBERMAN, H.; FRY, C. ZStep 95: a reversible, animated source code stepper. In: STASKO, J. T.; DOMINGUE, J. B.; BROWN, M. H.; PRICE, B. A. (Ed.). **Software visualization**: programming as a multimedia experience. [Cambridge, MA]: MIT Press, 1998. Chap. 19. p. 277-292.

LYLE, J. R.; WEISER, M. Automatic program bug location by program slicing. In: INTERNATIONAL CONFERENCE ON COMPUTERS AND APPLICATIONS, 2., 1987, Peking. **Proceedings**. Los Alamitos: IEEE, 1987. p. 877-882.

MALDONADO, J. C.; CHAIM, M. L.; JINO, M. Bridging the gap in the presence of infeasible paths: potential uses testing criteria. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SOCIETY, 12., 1992, Santiago. **Proceedings**... [Santiago: s.n., 1992a]. p. 323-340.

MALDONADO, J. C.; FABBRI, S. C. P.F. Verificação e validação de software. In: ROCHA, A. R.C.; MALDONADO, J. C.; WEBER, K. C. (Ed.). **Qualidade de software**: teoria e prática. São Paulo: Prentice-Hall, 2001. p. 66-73.

MARX, D. I. S.; FRANKL, F. G. Path-sensitive alias analysis for data flow testing. **Software Testing, Verification and Reliability,** v. 9, p. 51-73, 1999.

MARX, D. I. S.; FRANKL, P. G. The path-wise approach to data flow testing with pointer variables. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 1996, San Diego. **Proceedings**... New York: ACM Press, 1996. p. 135-146.

MYERS, G. J. **The art of software testing**. New York: John Wiley, 1979. 177 p. (Business Data Processing, a Wiley Series).

NISHIMATSU, A.; JIHIRA, M.; KUSIMOTO, S.; INOUE, K. Call-Mark Slicing: an efficient and economical way of reducing slice. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 21., 1999, Los Angeles. **Proceedings**... Los Alamitos: IEEE Computer Society, 1999. p. 423-431.

NTAFOS, S. A data flow oriented program testing strategy. **IEEE Transactions on Software Engineering**, v. 10, n. 6, p. 795-803, 1984.

OSTRAND, T.J.; BALCER, M. J. The category-partition method for specifying and generating functional tests. **Communications of the ACM**, v. 31, n. 6, p. 676-686, 1988.

OSTRAND, T. J.; WEYUKER, E.J. Data flow-based test adequacy analysis for languages with pointers. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING ANALYSIS, 1991, Victoria, Canada. **Proceedings of the symposium on testing, analysis, and verification**. New York: ACM Press, 1991. p. 87-97.

PAN, H. **Software debugging with dynamic instrumentation and test-based knowledge**. 1993. 148 f. Thesis (Doctor of Philosophy) – Purdue University, West Lafayette.

PAN, H.; SPAFFORD, E. H. Toward automatic localization of software faults. In: PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE, 10., 1992, Portland. **Proceedings**... [Portland: s.n., 1992]. p. 92-209.

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. London: McGraw-Hill, 1994.

RAPPS, S.; WEYUKER, E.J. Selecting software test data using data flow information. **IEEE Transactions on Software Engineering**, v. 14, n. 4, p. 367-375, 1985.

ROSENBLUM, D. S. A practical approach to programming with assertions. **IEEE Transactions on Software Engineering**, v. 21, n. 1, p. 19-31, 1995.

SHAPIRO, E. Y. **Algorithmic program debugging**. Cambridge: MIT Press, 1983.

SHIMOMURA, T. Critical slice-based fault localization for any type of error. **IEICE Transactions on Information and Systems**, v. E76-D, n. 6, p. 656-667, 1993.

SHIMOMURA, T.; OKI, Y.; CHIKARAISHI, T.; OHTA, T. An algorithmic fault-locating method for procedural languages and its implementation FIND. In: INTERNATIONAL WORKSHOP ON AUTOMATED AND ALGORITHMIC DEBUGGINGÄ– AADEBUG´95, 2., 1995, Saint-Malo. **Proceedings**... Rennes: IRISA, 1995. p. 191-203.

SILVA, J. B. **PROTESTE+**: ambiente de validação automática da qualidade de software através de técnicas de teste e de métricas de complexidade. 1995. Dissertação (Mestrado) – CPGCC, Universidade Federal do Rio Grande do Sul, Porto Alegre.

STAA, A.von. **Programação modular**. Rio de Janeiro: Campus, 2000. 700 p.

STAA, A. von. Instrumentação. In: ROCHA, A. R.C.; MALDONADO, J. C.; WEBER, K. C. (Ed.). **Qualidade de software**: teoria e prática. São Paulo: Prentice-Hall, 2001. p. 226-237.

STALLMAN, R. M.; PESCH, R. H. **Debugging with GDB**: the GNU source-level debugger. 7. ed. Cambridge, MA: Free Software Foundation, 1999.

TEMPLER, K. S.; JEFFERY, C. L. A configurable automatic instrumentation tool for ANSI C. In: AUTOMATED SOFTWARE ENGINEERING CONFERENCE, 13., 1998, Honolulu. **Proceedings**... [Los Alamitos]: IEEE Computer Society, [1998]. p. 249.

TIP, F. A survey of program slicing techniques. **Journal of Programming Languages**, v. 3, p. 121-189, 1995.

TOLMACH, A.; APPEL, A. W. A debugger for standard ML. **Journal of Functional Programming**, v. 5, n. 2, p. 155-200, Apr. 1995.

URAL, H.; YANG, B. A structural test selection criterion. **Information Processing Letters**, v. 57-163, 1988.

VESSEY, I. Expertise in debugging computer program: a process analysis. **International Journal on Man-Machine Studies**, v. 23, n. 5, p. 459-494, Nov. 1985.

VIRAVAN, C. **Enhancing debugging technology**. 1994. Thesis (Doctor of Philosophy) - Purdue University, West Lafayette.

WEISER, M. Program slicing. **IEEE Transactions on Software Engineering**, v. 10, n. 4, p. 352-357, 1984.

WONG, W. E.; GOKHALE, S.; HORGAN, J. R.; TRIVEDI, K. S. Locating program features using execution slices. In: IEEE SYMPOSIUM ON APPLICATION – SPECIFIC SYSTEMS AND SOFTWARE ENGINEERING AND TECHNOLOGY, 1999, Richardson. **Proceedings**... [Los Alamitos]: IEEE Computer Society, [1999]. p. 194.

Embrapa
Informática Agropecuária