Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Depuração de Programas Baseada em Informação de Teste Estrutural

Tese de Doutorado

Autor: **Marcos Lordello Chaim**

Orientador: **Prof. Dr. Mario Jino**
Co-orientador: **Prof. Dr. José Carlos Maldonado**

Banca Examinadora: **Prof. Dr. Adalberto Nobiato Crespo**
Profa. Dra. Ana Cristina Vieira de Melo
Prof. Dr. Eleri Cardozo
Prof. Dr. Ivan Luiz Marques Ricarte
Prof. Dr. José Raimundo de Oliveira
Prof. Dr. Manoel Gomes de Mendonça Neto

Tese submetida à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, para preenchimento dos pré-requisitos parciais para obtenção do Título de Doutor em Engenharia Elétrica.

Novembro de 2001

Resumo

Teste e depuração são duas atividades que têm um impacto importante no custo e na qualidade do produto gerado durante o processo de software. Estima-se que os esforços para detecção da presença (teste) e subseqüente localização e correção (depuração) de defeitos consomem entre 50% a 80% do orçamento de desenvolvimento e manutenção. O uso na depuração de informação gerada durante o teste tem sido preconizado com o objetivo de melhorar a qualidade do software e reduzir os custos de ambas as atividades.

Os requisitos de teste de critérios estruturais têm sido utilizados na depuração de programas para a identificação de um trecho de código com alta probabilidade de conter um defeito. Heurísticas são usadas para selecionar dentre os requisitos exercitados durante o teste aqueles mais relacionados com a ocorrência de falhas; os comandos associados a esses requisitos são candidatos a conter o defeito. Um dos problemas da utilização de informação de teste estrutural dessa maneira é que o trecho de código não necessariamente contém o defeito, dado que heurísticas foram utilizadas para selecionar os requisitos de teste.

Todavia, as indicações úteis para a localização do defeito fornecidas pelos requisitos selecionados não se restringem aos comandos associados a eles. Outras indicações, obtidas em tempo de execução, podem auxiliar o mantenedor de software a localizar o defeito. Para capturar essas indicações, é proposto o conceito de *requisito de teste revelador de erro*, entendendo-se por *erro* o estado intermediário caracterizado por um comportamento incorreto ou um desvio da especificação. Um estudo empírico foi realizado para avaliar a habilidade das heurísticas em selecionar requisitos reveladores de erro. Os resultados do estudo indicam que as heurísticas selecionam requisitos reveladores de erro mesmo quando não possuem habilidade para selecionar um trecho de código que contém o defeito.

Uma *estratégia de* **D***epuração baseada em* **R***equisitos de* **T***este* (**DRT**), para auxiliar o mantenedor a localizar o defeito nessas situações, é proposta e implementada. A estratégia **DRT** baseia-se: i) no uso de heurísticas para identificação de um conjunto inicial de requisitos candidatos a revelar erros; e ii) em dois mecanismos para auxiliar o mantenedor a identificar sucessivamente requisitos reveladores de erro até a localização do defeito. Duas ferramentas — `pokedebugheur` e `gdb/poke` — foram especificadas e desenvolvidas para apoiar a aplicação dessa estratégia. Ambas as ferramentas utilizam informação já coletada durante o teste estrutural e implementam algoritmos de baixo custo, quando comparados com os algoritmos implementados em estratégias similares. O alto custo dessas estratégias tem inviabilizado a sua aplicação em programas reais; a estratégia **DRT** pode ser uma alternativa escalável a elas.

Abstract

Testing and debugging are two activities that impact significantly the cost and quality of the product generated in the software process. The efforts for detection (testing) and for localization and correction (debugging) of faults consume between 50% and 80% of the development and maintenance budget. The use of information generated during the test for debugging purposes has been proposed as a means of improving the quality of the software and of reducing such costs.

Testing requirements of structural criteria have been utilized in program debugging to identify a piece of code with a high probability of including a fault. Heuristics have been used to select from the testing requirements exercised during testing those most related to the occurrence of failures; the program statements associated to these requirements are candidates to include the fault. One problem of using testing information in this way is that the piece of code does not necessarily contain the fault, since heuristics were used to select the testing requirements.

However, indications useful for fault localization provided by the testing requirements are not restricted to the associated statements. Other indications obtained at run-time can help the software maintainer in the fault localization task. To capture such indications, we propose the concept of *error-revealing testing requirement*, where *error* is the intermediate state of the program characterized by an incorrect behavior or a deviation from the specification. An empirical study was carried out to assess the heuristics ability in selecting error-revealing testing requirements. The results indicate that they do select error-revealing testing requirements even when they are not able to select a piece of code including the fault.

A **D***ebugging strategy based on* **R***equirements of* **T***esting* (**DRT**), to support the maintainer in such situations, is proposed and implemented. The **DRT** strategy is based on: i) the use of heuristics to identify an initial set of testing requirements which are candidates to reveal errors; and ii) two mechanisms to help the maintainer to successively identify error-revealing testing requirements until the fault localization. Two tools — `pokedebugheur` and `gdb/poke` — were specified and implemented to support the application of the new strategy. Both tools make use of information collected during structural testing and implement low cost algorithms in comparison to algorithms implemented in similar strategies. The high cost of such strategies has precluded their application in real programs; the **DRT** strategy may be a scalable alternative to them.

Agradecimentos

Agradeço ao meu orientador, Prof. Dr. Mario Jino, não somente por fornecer a orientação e as condições fundamentais para a realização desse trabalho, mas também por ter despertado em mim o interesse pela pesquisa científica e pela Engenharia de Software. Ao meu co-orientador, Prof. Dr. José Carlos Maldonado, pela orientação, estímulo e amizade.

Ao prof. Dr. Aditya P. Mathur, por me receber no *Software Engineering Research Center* (SERC) da Universidade Purdue durante o meu período de doutorado sanduíche. Aos meus amigos do SERC: João Cangussu, Baskar, Ramkumar, Prya, Sudipto e Sambrama. Ao Prof. Dr. Eric Wong da Universidade do Texas em Dallas, por fornecer os programas e os dados utilizados nos experimentos dessa tese.

Ao meus amigos do grupo de teste de software do DCA/FEEC/Unicamp: Adriana, as gêmeas Adriana e Andréia, Cristina, Daniela, Dino, Paulinho, Plínio Leitão, Plínio Vilela e Virgínia. Ao amigos do LCA: Armando e Myriam; Ricardo e Luciana; e Luiz Gonzaga.

À Embrapa Informática Agropecuária, na pessoa dos meus ex-chefes, Álvaro Seixas e Moacir Pedroso Júnior, e na pessoa do atual chefe, José Gilberto Jardine, pelo apoio profissional e financeiro. À CAPES e à Pró-reitoria de Pós-Graduação da Unicamp pelo apoio financeiro durante minha visita à Universidade Purdue.

Ao meu amigo, conselheiro e procurador José Ruy Porto de Carvalho, cuja imensa generosidade eu jamais vou ser capaz de retribuir. Aos meus amigos da Embrapa Informática Agropecuária; em especial, Carla, Roberto e Sílvio.

À minha esposa, Cristina, meu *Norte* e fonte de inspiração. Aos meus pais Aziz e Elza, que sempre estão ao meu lado, não importando o tamanho da dor e da dificuldade. Aos meus irmãos Tó, Baca e Cassinha, por terem feito a parte que me cabia, muito melhor do que eu seria capaz de fazer.

Para terminar, um agradecimento inusitado. Gostaria de agradecer a esses artistas maravilhosos, Caetano Veloso, Cidade Negra, Djavan, Fala Mansa, Jorge Aragão, Paulinho da Viola, Paralamas do Sucesso, Skank, Terra Samba, Zeca Pagodinho, entre outros, por manter minha Brasil-dependência em níveis aceitáveis durante meus períodos no exterior.

Para Aziz e Elza, que me deram quase tudo.
Para Cristina, que me deu o resto.

Sumário

Lista de Figuras

Lista de Tabelas

Notação e Abreviaturas

adpu Associação Definição-Potencial-Uso

adu Associação Definição-Uso

C Critério de teste estrutural

C_1 Custo de geração de casos de teste

C_2 Custo de execução de casos de teste

C_3 Custo de validação de casos de teste

C_4 Custo de análise de adequação

C_5 Custo de análise de não-executabilidade

$CE_{(H,C,T_i)}$
: Conjunto de requisitos candidatos a reveladores de erros selecionados pela aplicação de (H, C) no conjunto T_i

$CE_{\text{médio}}$
: Tamanho médio dos conjuntos $CE_{(H,C,T_i)}$ que incluem pelo menos um *rt-re*

$Cmd_{(H,C,T_i)}$
: Conjunto de comandos associados aos requisitos r pertencentes a $CE_{(H,C,T_i)}$

$Cmd_{\text{médio}}$
: Tamanho médio dos conjuntos $Cmd_{(H,C,T_i)}$ que incluem o sítio do defeito

$\text{Custo}_{(H,C)}$
: Custo empírico de aplicação do par (H, C) em conjuntos de casos de teste T_i, $1 < i \leq n$

D Conjunto de variáveis definidas em um nó *i*

DDT Paradigma de Depuração Depois do Teste

des Desvio Padrão

DRT Estratégia de **D**epuração Baseada em **R**equisitos de **T**este.

$E_{(C,t)}$ Conjunto de requisitos reveladores de erro exercitados por um caso de teste $t \in T_R$

$E_{(C,T_i)}$
Conjunto de requisitos de teste reveladores de erros exercitados por casos de teste reveladores de defeito t pertencentes a T_i

EF Eficácia de um conjunto de casos de teste T adequado a um critério C cujos requisitos foram determinados utilizando um dado modelo de dados

Ef_{cmd}
Eficiência em termos de comandos

Ef_{erro}
Eficiência em termos de requisitos reveladores de erros

G Grafo de fluxo de controle que representa o programa P

GT-C Cenário Grupo de Teste

H Heurística para seleção de requisitos de teste estrutural candidatos a revelar erros

H1 Heurística que seleciona os requisitos do conjunto $\cap RT_C(t)$ como *rt-c-re*s

H2 Heurística que seleciona os requisitos do conjunto $RT_C(t) - RT_C(t')$, onde $t \in T_R$ e $t' \in T_{NR}$, como *rt-c-re*s

H3 Heurística que seleciona os requisitos cujos valores de peso p_r foram classificados em primeiro lugar como *rt-c-re*s

H4 Heurística tipo *ranking* que seleciona os requisitos cujos valores p_r foram classificados em primeiro e segundo lugares como *rt-c-re*s

Inc_{erro}
Taxa de inclusão de requisitos reveladores de erro

$Inc_{\text{sítio}}$
Taxa de inclusão do sítio do defeito

$MC_C(T)$
Medida de cobertura obtida a partir da análise de adequação do conjunto de casos de teste T com respeito ao critério de teste estrutural C

med Valor Médio

MT-C
Cenário de Manutenção

N Conjunto de blocos de comandos (*nós*) de um programa P

P Programa (procedimento ou função) que faz parte de um software $\mathcal{S}$

p_r Peso atribuído ao requisito de teste r pelas heurísticas H3 e H4

R Conjunto de ramos do grafo de fluxo de controle G do programa P

refine_branch
: Novos requisitos a serem investigados obtidos a partir da situação reveladora de erros *Alcance Incorreto*

refine_use
: Novos requisitos a serem investigados obtidos a partir das situações reveladoras de erros *Valor Incorreto* e *Valor de Retorno Incorreto*

RT_C Conjunto de requisitos de teste estabelecido por um critério estrutural C

rt-c-re
: Requisito de Teste Candidato a Revelar Erros

rt-c-re$_{\text{Alcance}}$
: Requisitos de teste candidatos a revelar erros selecionados a partir da situação *Alcance Incorreto* (**refine_branch**)

rt-c-re$_{INT}$
: Requisitos de teste de integração candidatos a revelar erros selecionados a partir das situações reveladoras de erros *Valor Incorreto* e *Valor de Retorno Incorreto* (**refine_use**)

rt-c-re$_{\text{Valor}}$
: Requisitos de teste de unidade candidatos a revelar erros selecionados a partir das situações reveladoras de erros *Valor Incorreto* e *Valor de Retorno Incorreto* (**refine_use**)

$RT_C(t)$
: Conjunto de requisitos de teste estabelecido por um critério estrutural C exercitados pelo caso de teste $t \in T$

rt-re Requisito de Teste Revelador de Erro

S Comando de um programa P onde está localizado o sítio do defeito

sel(r) Conjunto de requisitos selecionados utilizando **refine_use** ou **refine_branch** a partir do requisito revelador de erro r

T Conjunto de casos de teste

Tam	Tamanho (número de de casos de teste) de um conjunto T adequado a um critério C cujos requisitos foram determinados utilizando um dado modelo de dados
T_i	I-ésimo conjunto de casos de teste
T_{NR}	Conjunto de casos de teste não-reveladores de defeito
TP-C	Cenário Testador-Programador
T_R	Conjunto de casos de teste reveladores de defeito
T_{R_r}	Conjunto de casos de teste reveladores de defeito que exercitam um requisito r em particular
T_{NR_r}	Conjunto de casos de teste não-reveladores de defeito que exercitam um requisito r em particular
V	Conjunto de variáveis que possuem valores incorretos em uma instância i^p

Capítulo 1

Introdução

> *Let software disappear, and life as we know it would break down, at least in developed countries. It controls most of the objects which surround us: computers, of course, but also telephones, cars, toys, tvs, much of our transportation system, and so on. Yet if the vision of web services comes to pass, today's dependence on software will appear slight. Life in the cloud (the world wide web) will mean that much of what we do, as* homo oeconomicus *at least, will be automated, from restaurant reservations to car purchases, from share trades to entire business deals.*
>
> *All this is at least some years off, and may not happen at all. But the prospect raises some interesting questions. Who will write all code needed for these services? What needs to be done to ensure that it is reliabe and secure?*
>
> (Survey: Software, *The Economist*, Abril, 2001)

1.1 Contexto

Neste início de novo milênio, é trivial observar que o uso de sistemas baseados em computador é prevalente em todas as atividades humanas. No entanto, um fenômeno muito recente acentuou, de forma dramática, o uso e a dependência dos sistemas computacionais. Um novo setor da economia — chamado de *nova economia* — está totalmente centrado na interconexão de computadores via rede. Na nova economia, atividades humanas básicas como o comércio e o ensino ocorrem no espaço digital estabelecido por computadores interconectados.

O *motor* desse espaço econômico digital é o *software*, que *move* não somente os próprios computadores, mas estabelece a suas interconexões, ou seja, a própria rede. Daí a importância cada vez maior da disciplina de Engenharia de Software cujo objetivo é propor métodos, ferramentas e procedimentos para aumentar a produção de software e, sobretudo, a sua qualidade. Neste sentido, segundo Osterweil et al. (1996), a qualidade de software será o critério dominante de sucesso na indústria de software. *Qualidade de software* é definida por Pressman (1994) como: i) requisitos funcionais e de desempenho estabelecidos; ii) padrões de desenvolvimento explicitamente documentados; e iii) características ímplicitas que são esperadas de todo software desenvolvido profissionalmente. Rocha (2001), por sua

vez, coloca esse conceito de forma mais resumida, porém similar: "Qualidade de sofware pode ser vista como um conjunto de características que devem ser alcançadas em um determinado grau para que o produto atenda às necessidades de seus usuários".

A qualidade é obtida durante todo o processo de software, sendo que várias atividades devem ser conduzidas para a obtenção de produtos com essa característica. Entre essas atividades destacam-se a *especificação* e *verificação* utilizando técnicas formais e rigorosas, as *revisões* e o *teste* de software (Maldonado, 1991). Portanto, não é uma única ação que confere qualidade ao produto, mas sim ações coordenadas dentro de um processo. Para exemplicar este aspecto, as evidências indicam que as revisões e o teste são atividades complementares (Maldonado e Fabbri, 2001), de forma que um processo de software de qualidade não pode prescindir de realizar ambas.

O *teste*, em especial, é um elemento usado para fornecer evidências da confiabilidade do software em complemento a outras atividades, sendo relevante para a identificação e eliminação de defeitos que persistem no software (Maldonado, 1991). Seu objetivo é revelar a presença de defeitos. No entanto, uma vez detectada a presença dos defeitos, eles precisam ser localizados e removidos, sem o que a qualidade do software não é aumentada. A *depuração* de software tem por objetivo realizar estas duas tarefas. O contexto desta tese envolve essas duas atividades distintas, porém muito relacionadas, do processo de software.

1.1.1 Teste de Software

O objetivo da atividade de teste é detectar a presença de defeitos que, porventura, tenham passado desapercebidos durante a fase de desenvolvimento. Para tanto, casos de teste são desenvolvidos e submetidos ao programa com o objetivo de produzir uma saída que esteja em desacordo com a especificação. Quando isto ocorre, diz-se que o teste foi bem sucedido (Myers, 1979). Caso defeitos não sejam detectados, o testador tem aumentada a sua confiança quanto à qualidade do software. Porém, para que isto ocorra, é preciso que os testes tenham sido realizados de maneira rigorosa e sistemática.

Nas últimas três décadas, houve um grande esforço de pesquisa para o desenvolvimento de estratégias, técnicas, critérios e ferramentas de teste. Como resultado, o teste passou de uma atividade *ad hoc*, considerada quase que uma arte (Myers, 1979), para um ramo da Engenharia de Software dotado de técnicas sistemáticas (Delamaro, 1997; Delamaro et al., 2001a; DeMillo et al., 1978; Fabbri, 1996; Harrold e Soffa, 1991; Herman, 1976; Jin e Offutt, 1998; Laski e Korel, 1983; Linnenkugel e Müllerburg, 1990; Maldonado, 1991; Maldonado et al., 1992a; Myers, 1979; Ntafos, 1984; Ostrand e Balcer, 1988; Rapps e Weyuker, 1985; Souza, 2000; Vergilio, 1997; Vilela, 1998; Ural e Yang, 1988) baseadas em fundamentos teóricos (Frankl e Weyuker, 1993; Goodenough e Gerhart, 1975; Gutjahr, 1999; Wah, 2000) e apoiadas por ferramentas automatizadas (Chaim, 1991; Delamaro, 1993; Delamaro e Maldonado, 1996; Delamaro, 1997; DeMillo et al., 1988; Frankl e Weyuker, 1988; Horgan e London, 1991; Maldonado et al., 1992b; Offutt et al., 2000; Ostrand e Weyuker, 1991; Silva, 1995).

As técnicas de teste são classificadas em *estruturais*, *funcionais*, *baseadas em defeitos* e *baseadas em máquinas de estados finitos*, de acordo com a origem da informação de teste utilizada para estabelecer os *requisitos de teste*. Essas técnicas contemplam diferentes perspectivas do software de forma que deve-se explorar suas vantagens e seus aspectos complementares ao se estabelecer uma estratégia de teste (Maldonado e Fabbri, 2001).

A técnica estrutural utiliza a implementação para determinar requisitos que devem ser exercitados pelos casos de teste. Já a técnica funcional estabelece os requisitos com base na especificação do software. A técnica baseada em defeitos, por sua vez, visa derivar casos de teste que mostrem a presença ou a ausência de defeitos mais comuns. A técnica de teste com base em máquinas de estados finitos utiliza a estrutura de máquinas de estado finito e o conhecimento subjacente para derivar requisitos de teste. Em geral, as técnicas acima podem ser utilizadas tanto para o teste de unidade (teste de cada unidade isoladamente) como para o teste de integração (teste da interação entre as unidades que compõem o software).

A atividade de teste possui limitações que independem da técnica utilizada. Segundo Maldonado e Fabbri (2001), os seguintes problemas relacionados com o teste são, em geral, indecidíveis: "dados dois programas, se eles são equivalentes; dadas duas seqüência de comandos (caminhos) de um programa, ou de programas diferentes, se elas computam a mesma função, e dado um caminho, se ele é executável ou não, ou seja, se existe um conjunto de dados de entrada que leve à execução desse caminhos". Outras limitações do teste são a ocorrência de correção coincidente (um programa produz a saída esperada para um particular dado que exercita um determinado requisito; porém, se outro dado fosse escolhido, o resultado seria incorreto) e a impossiblidade de testar todas as entradas possíveis de um programa (normalmente o domínio dos dados de entrada é muito grande).

Nesse contexto, os requisitos definidos pelas técnicas de teste estabelecem *critérios de teste* cujo objetivo é auxiliar o testador a responder duas perguntas: como devem ser selecionados os dados de teste? e como decidir se um produto foi suficientemente testado? (Maldonado e Fabbri, 2001) Em outras palavras, os critérios de teste podem ser utilizados para auxiliar a *seleção* de casos de teste e a análise da *adequação* de um conjunto de casos de teste já existente.

Com relação à análise de adequação, um conjunto de casos de teste T é adequado a um critério de teste C se todos os requisitos estabelecidos pelo critério são exercitados pelos elementos de T. Por exemplo, os critérios de teste estruturais podem derivar seus requisitos a partir do fluxo de controle (Myers, 1979) ou a partir do fluxo de dados do programa (Herman, 1976). Um exemplo de critério de teste de fluxo de controle é o critério *todos ramos*. Um conjunto de casos de teste é adequado a este critério se incluir casos de teste que executam (exercitam), pelo menos uma vez, todas as possíveis saídas dos comandos de controle de fluxo condicional (e.g., *if*, *while*, *for*, *case* etc). Já o critério de teste de fluxo de dados *todos usos* (Rapps e Weyuker, 1985) requer que os conjuntos adequados a ele exercitem pelo menos um caminho livre de definição entre a definição (ponto do programa onde um valor é atribuído) e os subseqüentes usos (referências ao valor atribuído) de uma variável.

Os critérios baseados em análise de fluxo de dados para serem aplicados em programas reais requerem o estabelecimento de um *modelo de dados* que define como os conceitos de definição e uso de uma variável são tratados com relação aos recursos mais comuns das linguagens de programação (e.g., vetores, ponteiros, registros etc). O modelo de dados estabelece o nível de precisão da análise de fluxo de dados utilizada para determinar os requisitos de teste. Por exemplo, os requisitos podem ser determinados considerando-se ou não os fluxos relativos ao uso de ponteiros. Quando considerados, esses fluxos podem ser identificados estaticamente (Horgan e London, 1991; Vilela et al., 1997) ou dinamicamente (Ostrand e Weyuker, 1991). Marx e Frankl (1996, 1999) argumentam que a análise de fluxo de dados mais precisa pode facilitar a detecção de alguns tipos especiais de defeitos como aqueles relacionados com o uso incorreto de ponteiros a um custo razoável.

Os critérios estabelecidos pelas técnicas de teste têm sido avaliados e comparados teoricamente (Frankl e Weyuker, 1993; Vilela, 1998) e empiricamente (Delamaro et al., 2001a; Frankl e Weiss, 1993; Frankl et al., 1997; Frankl e Iakounenko, 1998; Hutchins et al., 1994; Mathur e Wong, 1993; Wong, 1993) quanto à *eficácia* dos conjuntos de casos de teste adequados a eles. Entende-se por eficácia a probabilidade de um conjunto adequado a um critério de teste detectar os defeitos presentes no software (Frankl et al., 1997). De maneira geral, os estudos empíricos (Delamaro et al., 2001a; Frankl e Weiss, 1993; Frankl et al., 1997; Frankl e Iakounenko, 1998; Hutchins et al., 1994; Souza, 1996; Vergilio et al., 1996) indicam que os conjuntos de casos de teste derivados utilizando técnicas de teste são mais eficazes na detecção dos defeitos do que conjuntos de casos de teste derivados aleatoriamente (teste aletório). Resultados teóricos recentes corroboram esta indicação (Gutjahr, 1999).

Portanto, os trabalhos desses pesquisadores indicam que o testador de software dispõe atualmente de um conjunto de técnicas de teste sistemáticas cuja eficácia na detecção de defeitos tem sido mostrada experimentalmente e comprovada teoricamente. No entanto, o que se observa é que o uso dessas técnicas não é uma prática comum na indústria de software. E a principal razão para esta situação é que, apesar de eficazes, a aplicação das técnicas de teste é ainda custosa (Harrold, 2000) de forma que não existem evidências empíricas suficientes quanto ao custo/benefício dessas técnicas.

1.1.2 Depuração de Software

A depuração de software é comumente definida como a tarefa de *localização* e *remoção* de defeitos (Araki et al., 1991). Normalmente, ela é entendida como um corolário do teste bem sucedido (Agrawal et al., 1995), ou seja, ela ocorre sempre que um defeito é revelado. No entanto, defeitos podem ser revelados em diferentes fases do ciclo de vida de um software, sendo que a depuração possui características diferentes dependendo da fase em que se encontra o software. Por isso, os processos de depuração que ocorrem *durante a codificação*, *depois do teste* e *durante a manutenção* são distinguidos.

A depuração durante a codificação é uma atividade complementar à de codificação. Já a depuração depois do teste é ativada pelo teste sistemático bem sucedido, podendo beneficiar-se da informação

coletada durante aquela fase. A depuração durante a manutenção, por sua vez, acontece devido a uma necessidade de manutenção no software, que pode ter sido causada, por exemplo, por um defeito revelado depois de liberado o software ou pela necessidade de acrescentar novas características a ele.

A relevância da atividade de depuração tem dirigido esforços de pesquisa em duas direções: no *entendimento* do processo de depuração (Araki et al., 1991; Luckey, 1980; Vessey, 1985) e no *desenvolvimento* de técnicas que aumentem a sua produtividade (Agrawal e Horgan, 1990; Agrawal et al., 1995; Adams e Muchnick, 1986; Collofello e Cousins, 1987; Curcio, 1998; Fritzson et al., 1992; Korel e Laski, 1988; Korel, 1988; Lyle e Weiser, 1987; Pan e Spafford, 1992; Rosenblum, 1995; Stallman e Pesch, 1999; Shapiro, 1983; Weiser, 1984). Vários experimentos foram desenvolvidos com o objetivo de *entender* o processo de depuração de software e estabelecer um modelo de depuração. Araki et al. (1991), utilizando resultados de experimentos anteriores (Vessey, 1985), definiram o modelo de depuração chamado de *Hipótese-Validação* descrito na Figura 1.1.

Trata-se de um modelo cognitivo que caracteriza a atividade de depuração como um processo interativo de síntese, verificação (os termos *verificação* e *verificar* estão sendo utilizados no sentido de *confirmação*) e refinamento de hipóteses (Viravan, 1994). De acordo com este modelo, o responsável pela depuração do software — chamado a partir de agora de *mantenedor* visto que a atividade de depuração é essencialmente uma manutenção do programa já implementado — estabelece hipóteses com relação à localização do defeito e à modificação necessária para corrigir o programa. O processo de depuração é guiado pela verificação e refutação das hipóteses levantadas, bem como pela geração de novas hipóteses e refinamento das já existentes (Araki et al., 1991). Note-se que o modelo Hipótese-Verificação é genérico, não sendo vinculado a um tipo particular de depuração. Outros autores (Chan, 1997; DeMillo et al., 1996) também desenvolveram modelos de depuração que são variantes do modelo de Araki et al. (1991) orientados, porém, para técnicas específicas de depuração.

Técnicas e ferramentas de depuração propriamente ditas têm sido desenvolvidas para auxiliar a realização de uma ou mais tarefas descritas no modelo Hipótese-Validação. Um exemplo típico são os depuradores simbólicos (Adams e Muchnick, 1986; Stallman e Pesch, 1999). Este tipo de ferramenta permite que a execução do programa pare em pontos determinados pelo mantenedor e que os valores das variáveis e da pilha de execução sejam examinados. Neste sentido, eles apóiam a verificação e a refutação das hipóteses levantadas pelo mantenedor.

Outras técnicas, por sua vez, apóiam a geração de hipóteses a serem verificadas. Por exemplo, a inclusão de *asserções* (Curcio, 1998; Rosenblum, 1995; Staa, 2000, 2001) no código fonte do programa visa identificar pontos do programa que, ao serem atingidos durante a execução do caso de teste, violam a especificação original do software. Esses pontos indicam locais onde os efeitos do defeito se manifestam. A técnica de *fatiamento* de programas (Agrawal e Horgan, 1990; Korel e Laski, 1988; Weiser, 1984), por sua vez, identifica um conjunto de comandos (chamado de *fatia* do programa) candidatos a conter o defeito. A *fatia* é composta de comandos que afetam o valor de uma variável em determinado ponto do programa. Adicionalmente, a eleição de comandos candidatos a conter defeitos pode ser feita por

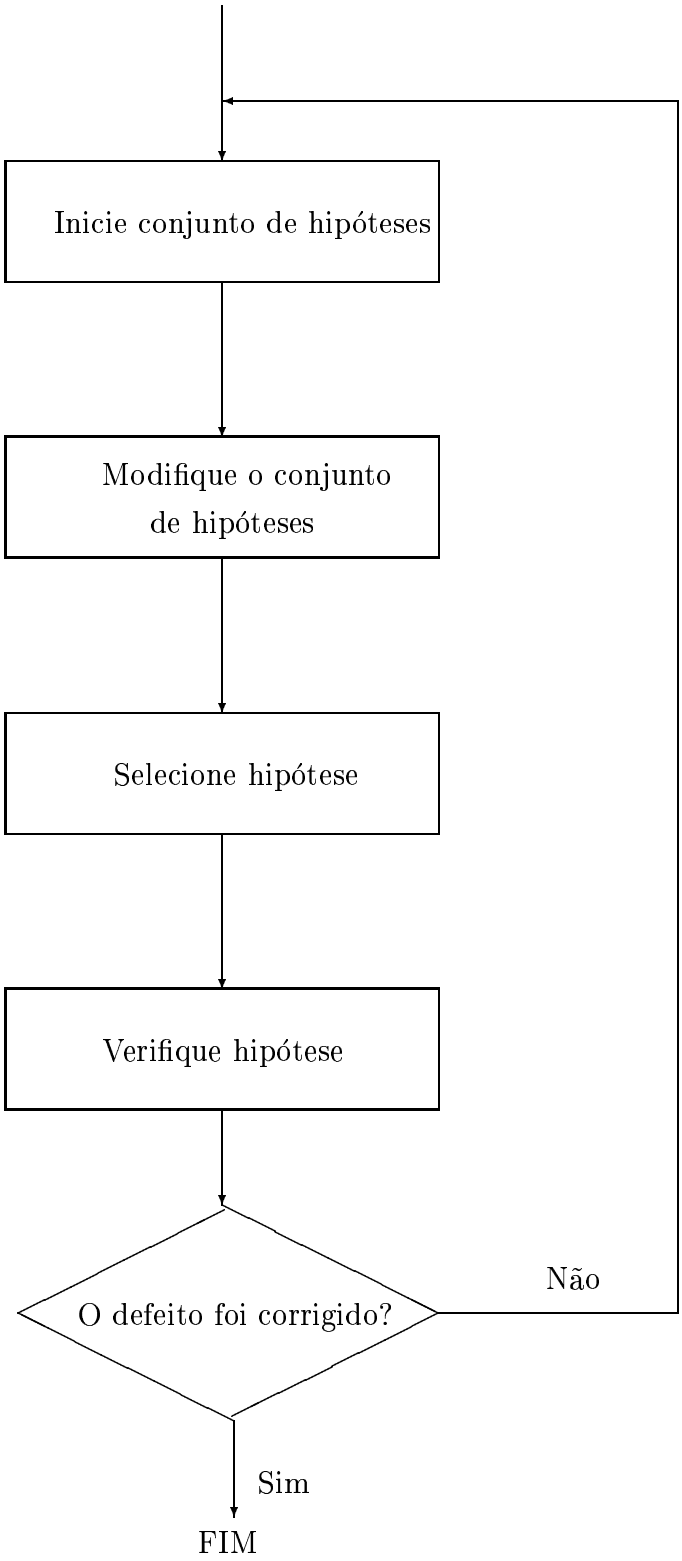


Figura 1.1: Modelo hipótese-validação para o processo de depuração (Araki et al., 1991).

meio de heurísticas que manipulam tanto *fatias* de programas como informação coletada durante o teste (Agrawal et al., 1995; Collofello e Cousins, 1987; Lyle e Weiser, 1987; Pan e Spafford, 1992). A técnica de depuração algorítmica (Fritzson et al., 1992; Korel, 1988; Shapiro, 1983), por sua vez, apóia duas tarefas do modelo Hipótese-Validação: geração e seleção de hipóteses. Ela apóia as duas tarefas guiando o mantenedor até o local do defeito utilizando uma busca binária em um conjuntos de comando suspeitos.

Como pode se observar, todas as tarefas do modelo Hipótese-Validação são apoiadas por técnicas e ferramentas. No entanto, em ambientes industriais de desenvolvimento de software, é comum o uso apenas de depuradores simbólicos para apoiar a depuração de software. Novamente, a causa para esta situação é o custo das técnicas de geração e seleção de hipóteses, que não são escaláveis para uso em sistemas reais (Lian et al., 1997; Nishimatsu et al., 1999; Wong et al., 1999).

A situação descrita é mais dramática quando se leva em consideração dois aspectos em relação aos depuradores simbólicos: (1) tratam-se de ferramentas disponíveis desde o final dos anos 60 que (2) basicamente apóiam apenas uma tarefa do modelo Hipótese-Validação — a verificação de hipóteses. Destas observações, conclue-se que a atividade de depuração é realizada da mesma maneira há pelo menos 30 anos e depende essencialmente da experiência do mantenedor, pois, para a geração e seleção de hipóteses, ele não conta com o apoio de técnicas e ferramentas.

1.2 Motivação

A motivação para a realização desse trabalho de pesquisa está no fato de que tanto as técnicas de teste sistemáticas (Harrold, 2000) como as de depuração mais modernas não têm sido adotadas em ambientes industriais devido ao seu custo de aplicação (Lian et al., 1997; Nishimatsu et al., 1999; Wong et al., 1999). O resultado final dessa situação é que tanto o teste como a depuração são conduzidos de forma inadequada, o que em última instância impacta a qualidade dos produtos de software distribuídos.

No entanto, se os resultados do teste puderem ser utilizados para auxiliar a depuração então é razoável alocar parte do custo do teste sistemático ao desta atividade. De maneira semelhante, as técnicas de depuração mais modernas podem ter seus custos reduzidos caso utilizem informação já disponível do teste. Isto é possível porque as informações coletadas pelas técnicas de depuração (e.g., *fatiamento* de programas, depuração algorítmica) são semelhantes às associadas aos requisitos de teste dos critérios definidos pelas técnicas estruturais e baseadas em defeitos.

Portanto, a conjectura que motiva o trabalho é que a *utilização* de informação de teste na depuração pode reduzir os custos de aplicação tanto do teste sistemático como das técnicas mais modernas de depuração em sistemas reais. A idéia é utilizar os requisitos exercitados pelos casos de teste para apoiar as tarefas do modelo Hipótese-Validação. Algumas iniciativas neste sentido já foram realizadas (Agrawal et al., 1995; Collofello e Cousins, 1987; DeMillo et al., 1996); porém, elas se restringem a uma das tarefas do modelo — a seleção do conjunto inicial de hipóteses. No entanto, para que o custo

possa ser realmente compartilhado entre as atividades, é necessário que *todas* as tarefas sejam apoiadas por informação de teste — seleção, verificação e geração de hipóteses. Como a informação coletada durante o teste é a base dessa abordagem, ela poderá ser utilizada na depuração que ocorre *depois* do teste e *durante* a manutenção, visto que os resultados do teste já estão disponíveis nesses pontos do processo de software.

1.3 Objetivos

O interesse principal dessa tese é investigar a conjectura de que a *utilização* de informação de teste na depuração pode reduzir os custos de aplicação tanto do teste sistemático como das técnicas mais modernas de depuração em sistemas reais. Em especial, este interesse está voltado para o uso dos requisitos estabelecidos por critérios de teste estruturais visto que há uma relação direta entre esses requisitos e a implementação do programa. Para realizar esta investigação, foram definidos os objetivos descritos abaixo:

- *Estudar as principais técnicas de depuração em relação ao tipo de informação de teste utilizada e a sua escalabilidade*. Para este estudo, utiliza-se como diretriz um modelo de depuração, derivado do modelo Hipótese-Validação, voltado especificamente para a depuração que ocorre *depois* do teste. Este modelo — chamado de paradigma de *Depuração depois do Teste* — enfatiza os seguintes aspectos: identificação, avaliação e refinamento sucessivo de sintomas internos até a localização do defeito; e tipo de informação de teste utilizada.

- *Investigar o custo e a eficácia do teste baseado em análise de fluxo de dados mais precisa na detecção de defeitos relacionados com o uso de ponteiros e campos de registros*. Do ponto de vista da depuração *depois* do teste, o nível da análise de fluxo de dados tem implicações importantes. Por um lado, é desejável que defeitos comuns e difíceis de detectar (como é o caso dos defeitos relativos ao uso incorreto de ponteiros e registros) sejam devidamente tratados durante o teste para que o processo de depuração seja ativado caso eles estejam presentes. Além disso, a análise de fluxo de dados mais precisa pode fornecer informações de teste adicionais úteis para a localização de defeitos desse tipo. Por outro lado, o custo de obtenção dessas informações deve ser razoável, pois, caso contrário, a escalabilidade das técnicas de depuração baseadas nelas poderá ser inviabilizada.

- *Avaliar a habilidade dos requisitos de teste em fornecer informações úteis que levem à localização do defeito*. Para que os requisitos de teste possam ser utilizados na depuração eles devem conter informações que auxiliem o mantenedor a localizar o defeito. Essas informações podem ser representadas por um trecho de código onde o defeito está contido ou um subconjunto de

requisitos de teste que fornecem indicações úteis, em tempo de execução, para a localização do defeito.

- *Propor uma estratégia de depuração baseada em informação de teste que apóie todas as tarefas de depuração*. Esta estratégia deve fornecer mecanismos para que o mantenedor possa localizar o defeito a partir das indicações fornecidas pelos requisitos de teste.

- *Desenvolver e implementar mecanismos eficientes para automatização da estratégia proposta*. Por eficiente, entende-se que estes mecanismos devem possuir baixo custo em tempo de depuração de forma a poderem vir a ser utilizados em aplicações reais.

Em resumo, os objetivos do trabalho são: *analisar* o uso de informação de teste na localização de defeitos; e *desenvolver* estratégias de depuração baseadas nesse tipo de informação completas e de baixo custo de aplicação.

1.4 Organização

Neste capítulo, foram discutidos o contexto, a motivação e os objetivos do trabalho de pesquisa realizado.

No Capítulo 2, são descritos os principais termos e técnicas de teste utilizados no decorrer da tese. Além disso, é realizada uma revisão das várias técnicas de depuração com relação ao paradigma de *Depuração depois do Teste* e quanto a sua escalabilidade.

No Capítulo 3, é investigado o teste baseado em análise de fluxo de dados mais precisa na detecção de defeitos relacionados com o uso incorreto de ponteiros e campos de registros. Dois modelos de dados mais precisos são definidos. A eficácia e o custo do teste de fluxo de dados utilizando estes modelos mais exigentes foram analisados por meio de um estudo de caso.

A habilidade dos requisitos de tese em fornecer informações úteis para localização de defeitos é analisada no Capítulo 4. Para esta análise, é proposto o conceito de *requisitos de teste reveladores de erro*. Este conceito engloba requisitos de teste que ou incluem o sítio do defeito no trecho de programa associado a eles ou fornecem indicações úteis para a localização do defeito em tempo de execução. Um estudo de caso foi conduzido para avaliar a habilidade dos requisitos de teste selecionados por meio de heurísticas em localizar o defeito propriamente dito ou identificar requisitos reveladores de erro.

No Capítulo 5, é proposta uma estratégia de depuração baseada na identificação sucessiva de requisitos de teste reveladores de erro. Esta estratégia é apoiada por dois mecanismos. O primeiro deles tem como objetivo auxiliar o mantenedor a identificar requisitos reveladores de erro. O segundo auxilia o mantenedor a selecionar *novos* requisitos a serem investigados a partir dos requisitos reveladores de erros identificados. Esta situação ocorre quando o requisito revelador de erro contém apenas indicações úteis à localização do defeito, mas não o sítio do defeito propriamente dito.

O Capítulo 6 apresenta os aspectos de implementação de uma ferramenta — `gdb/poke` — que implementa os mecanismos propostos no Capítulo 5. A principal característica dessa ferramenta é o seu baixo custo em tempo de depuração.

Finalmente, no Capítulo 7, são apresentadas as conclusões e as perspectivas de trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Neste capítulo, a depuração que ocorre *depois* do teste é analisada em detalhes. Inicialmente, são descritos os principais termos e técnicas de teste utilizados no decorrer da tese. Em seguida, várias técnicas de depuração são avaliadas em relação ao paradigma de *Depuração depois do Teste* e quanto a sua escalabilidade para programas reais. Este paradigma enfatiza os seguintes aspectos: identificação, avaliação e refinamento sucessivo de sintomas internos até a localização do defeito; e tipo de informação de teste utilizada.

2.1 Depuração de Programas no Processo de Software

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* (Adams e Muchnick, 1986; Stallman e Pesch, 1999). 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 (o defeito pode ainda estar localizado na interface do código antigo com o novo ou mesmo no código antigo, tendo sido revelado quando o novo foi introduzido). Outro instrumento importante para análise do novo código são as ferramentas que detectam usos inválidos de memória. Ferramentas comerciais como Purify (Hastings e 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 (Agrawal et al., 1995, 1998; Chen e Cheung, 1997; Collofello e Cousins, 1987; DeMillo et al., 1996; Fritzson et al., 1992; Korel e Laski, 1988; Lyle e Weiser, 1987; Weiser, 1984; Wong et al., 1999) 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 the 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 capítulo, a atividade de depuração que ocorre *depois* do teste é investigada em mais detalhes. Na Seção 2.2, são apresentadas as definições de alguns termos relacionados com as atividades de teste e depuração utilizados no decorrer do texto. A descrição das técnicas de teste cujos resultados podem ser utilizados na depuração está contida na Seção 2.3. Na Seção 2.4, é estabelecido o Paradigma de *Depuração depois do Teste* (DDT). Este paradigma 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 paradigma e a sua escalabilidade para sistemas reais na Seção 2.5. A Seção 2.6 contém as considerações finais.

2.2 Definição de Termos

As definições descritas a seguir são baseadas no padrão IEEE número 610.12-1990 (IEEE, 1991). No contexto desta tese, *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 Figura 2.1 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.

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 Figura 2.2 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}.

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 Figura 2.3. 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.

```
        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]);
        }
```

Figura 2.1: Programa exemplo 1.

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)`
$\vdots$	$\vdots$
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]);`

Figura 2.2: Execução do programa exemplo 1 com a entrada (i=5).

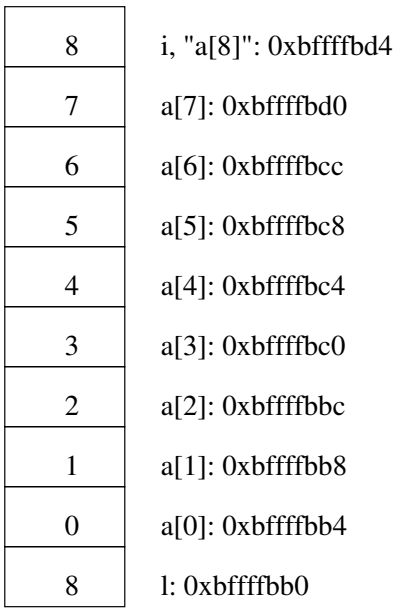


Figura 2.3: Estado da memória do programa depois do ponto de execuçao 5^{29}.

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.

2.3 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 (Myers, 1979; Ostrand e Balcer, 1988). 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 e Fabbri, 2001).

A seguir, as técnicas de teste estruturais são descritas em mais detalhes visto que seus resultados são utilizados nas técnicas de depuração desenvolvidas nesta tese.

2.3.1 Técnica Estrutural

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 $MC_C(T)$ 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 $MC_C(T)$ é dada pela relação abaixo:

$$MC_C(T) = \frac{|\bigcup RT_C(t)| \text{ para todo } t \in T}{|RT_C|}$$

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 $MC_C(T)$. A seguir, alguns critérios de teste estruturais são descritos.

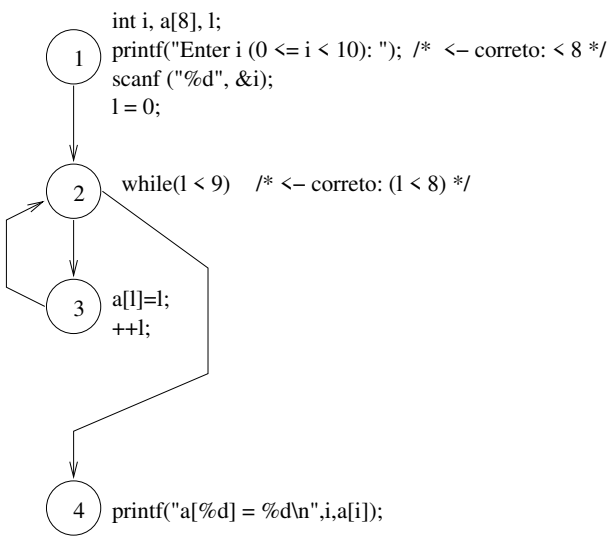


Figura 2.4: Grafo de fluxo de controle obtido do programa da Figura 2.1.

Conceitos de Fluxo de Controle e Fluxo de Dados

Os *conceitos básicos* de fluxo de controle e fluxo de dados, utilizados na definição dos requisitos de teste dos critérios estruturais, são introduzidos abaixo.

Seja P um programa escrito em uma linguagem do estilo Algol (Ghezzi e Jazayery, 1987) e S_1 ... S_i ... 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 Figura 2.4 apresenta o grafo de fluxo de controle obtido a partir do programa da Figura 2.1. 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 $(i, n_1, \ldots, n_m, j)$, $m \geq 0$, que não contenha definição de uma variável *x* nos nós $n_1, \ldots, n_m$ 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 $(n_1, n_2, \ldots, n_j, n_k)$ c.r.a. a uma variável *x*, onde

o caminho $(n_1, n_2, \ldots, n_j)$ é um caminho livre de laço e n_1 tem uma definição de *x*, é denominado *potencial-du-caminho c.r.a. x*.

Critérios Baseados em Fluxo de Controle

Os requisitos de teste destes critérios estão associados a elementos do fluxo de controle do programa.

- **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.
- **Critério Todos LCSAJs**: exige a execução de todos os LCSAJs presentes no programa. Um LCSAJ (*linear code sequence and jump*) (Hedley e Hennell, 1985) é definido como uma seqüência linear de código executável que começa ou no ponto inicial do programa ou em um ponto para o qual houve um desvio de fluxo de controle e termina com o comando final do programa ou com um comando de desvio de fluxo de controle.

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).

Critérios Baseados em Análise de Fluxo de Dados

Os critérios baseados em análise de fluxo de dados (chamados simplesmente de critérios de fluxo de dados) (Herman, 1976; Laski e Korel, 1983; Maldonado et al., 1992a; Ntafos, 1984; Rapps e Weyuker, 1985; Ural e Yang, 1988) 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 e Weyuker, 1988). Os critérios Potenciais Usos (Maldonado et al., 1992a) 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 e Weyuker, 1985) e Potenciais Usos (Maldonado et al., 1992a).

- **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,k) 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 (*i,j*), apenas que o nó *j* ou ramo (*i,j*) 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*.

Com o objetivo de reduzir o número de *adpu*s requeridas e, dessa maneira, reduzir o custo da análise de adequação dos critérios Potenciais Usos, Maldonado (Maldonado et al., 1992c) estendeu o conceito de *ramos essenciais* (Chusho, 1987). Os ramos essenciais constituem um subconjunto dos ramos do programa cuja propriedade é garantir a execução de *todos* os ramos do programa quando são executados. As *adpu*s foram então redefinidas utilizando a seguinte notação: (*i, (j,k), D*)), onde (*j,k*) é um ramo essencial estendido e D é um conjunto de variáveis definidas no nó *i*. Para satisfazer a *adpu* (*i, (j,k), D*), os casos de teste devem executar caminhos que alcancem (*j,k*) e que são livres de definição (potenciais du-caminhos) c.r.a. todas as variáveis $x \in D$. O exercício de todas as *adpu*s representadas utilizando ramos essenciais estendidos por casos de teste do conjunto T garante a adequação ao critério todos potenciais-usos (todos potenciais usos/du).

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 e Frankl, 1996, 1999) apóiam a aplicação dos critérios de Rapps e Weyuker para a linguagem Pascal; ATAC (Horgan e London, 1991) e Tactic (Ostrand e 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).

Caminhos Não-executáveis

Os critérios de teste estruturais, por serem baseados em caminhos, podem exigir que caminhos não-executáveis sejam exercitados. Isto ocorre porque a determinação dos requisitos de teste é realizada considerando-se apenas aspectos sintáticos do programa. Um caminho é *não-executável* se não existir um conjunto de valores de entrada, parâmetros e variáveis globais que provoquem a sua execução. Por exemplo, no programa da Figura 2.4, o caminho (1, 2, 4) é não-executável porque não existe dado de entrada que o execute.

É muito comum os requisitos de teste dos critérios estruturais exigirem caminhos não-executáveis, de maneira que é muito difícil obter valores de cobertura 100%. Uma solução é eliminar os requisitos de teste *não-executáveis*, isto é, aqueles para os quais não existem caminhos executáveis que os exercitem. Os critérios acima podem ser redefinidos para incluírem apenas requisitos de teste *executáveis* (Frankl e Weyuker, 1988). Para exemplificar estes novos critérios (referenciados como critérios*), é apresentada abaixo a definição dos critérios todos potenciais usos*:

Critério todos Potenciais Usos* : requer que todas as *adpu* dos tipos (*i, j,* x) e (*i, (j,k),* x) *executáveis* sejam exercitadas pelos casos de teste de um conjunto T.

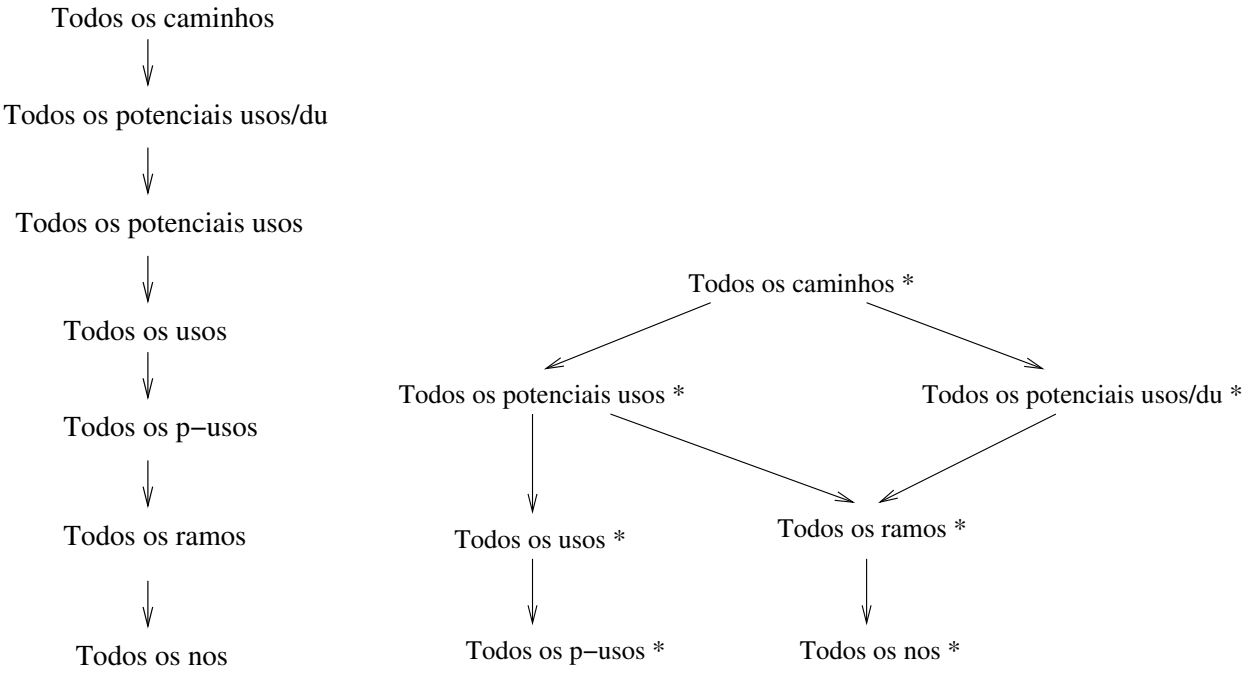
Apesar dos critérios* permitirem a obtenção de valores de coberutura de 100%, o cálculo dessa medida com respeito a esses critérios é, em geral, não-computável, pois a determinação da executabilidade de um requisito de teste é uma questão indecidível (Frankl e Weyuker, 1988).

No Apêndice E é descrito a determinação das *adpu*s baseadas em ramos essenciais estendidos considerando a presença de caminhos não-executáveis.

Comparação entre Critérios Estruturais

Os critérios de teste estruturais podem ser comparados teoricamente e empiricamente. Do ponto de visto teórico, uma das maneiras de se comparar dois critérios é por meio da relação de *inclusão* (Rapps e Weyuker, 1985; Frankl e Weyuker, 1988). Um critério C_1 *inclui* um critério C_2 se todo e qualquer conjunto de caminhos que *satisfaz* C_1 (i.e., exercita os requisitos de teste de C_1) também *satisfaz* C_2. Um critério C_1 *inclui estritamente* um critério C_2 se C_1 inclui C_2 e C_2 não inclui C_1. Dois critérios são *incomparáveis* se C_1 não inclui C_2 e C_2 não inclui C_1. A relação de inclusão indica o grau de exigência dos critérios. Em outras palavras, se C_1 inclui estritamente C_2, então mais caminhos são requeridos, mais casos de teste necessários, maior a exigência do critério. Na Figura 2.5, é apresentada a ordem parcial da relação de inclusão dos critérios definidos acima. Para o estabelecimento dessa hierarquia de critérios foram assumidas duas premissas: (1) todo nó de entrada contém pelo menos uma definição e (2) todo predicado contém pelo menos um p-uso (Vergilio, 1997).

Marré e Bertolino (1996) estabeleceram a relação de inclusão entre requisitos de teste. Um requisito de teste r_1 *inclui* outro requisito r_2 se qualquer caminho que exercita r_1 exercita também r_2.



(a) Ordem parcial dos critérios

(b) Ordem parcial dos critérios na presença de caminhos não-executátveis

Figura 2.5: Relação de Inclusão entre os Critérios Estruturais.

Do ponto de vista empírico, os critérios de teste estruturais podem ser comparados em termos de custo, de eficácia e de dificuldade de satisfação (Wong, 1993). O *custo* refere-se ao esforço necessário para utilizar o critério; a *eficácia* refere-se à probabilidade de um critério revelar um defeito (Frankl et al., 1997); e a *dificuldade de satisfação* refere-se à probabilidade de satisfazer um critério tendo satisfeito outro. A comparação empírica consiste na avaliação desses parâmetros em programas reais utilizando vários conjuntos de casos de teste diferentes adequados aos critérios.

2.4 Paradigma de Depuração Depois do Teste (DDT)

A seguir, várias técnicas de depuração de programas procedimentais são revisadas e analisadas. A análise é baseada no *Paradigma de Depuração Depois do Teste* (DDT). Este paradigma foi desenvolvido a partir do modelo Hipótese-Validação (Araki et al., 1991; Vessey, 1985); porém, sua ênfase está no uso de informação de teste em todas as tarefas de localização de defeitos. O paradigma DDT é descrito no algoritmo apresentado na Figura 2.6.

Embora o paradigma 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 paradigma 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 paradigma 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 (Subseção 2.3.1).

2.5 Técnicas de Depuração e o Paradigma DDT

Nesta seção, são analisadas várias técnicas de depuração com respeito ao paradigma DDT. As técnicas são apresentadas resumidamente e avaliadas com relação aos aspectos que o paradigma 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.

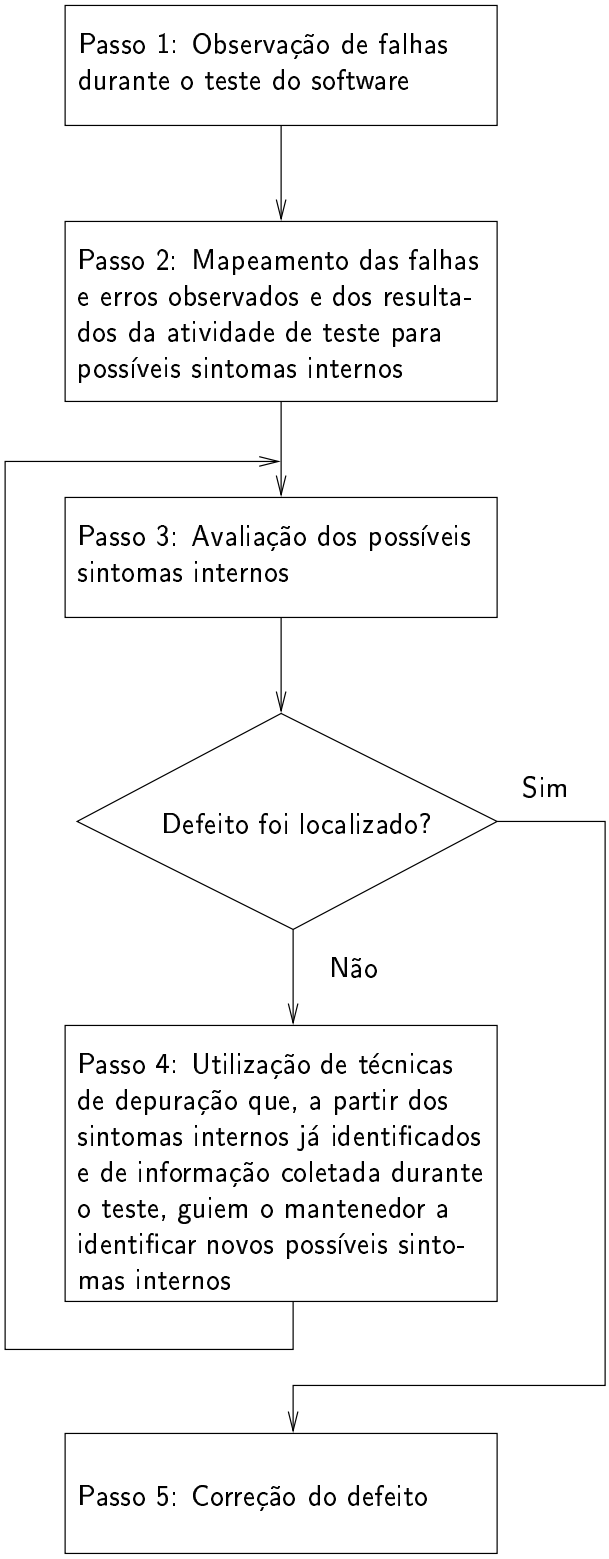


Figura 2.6: Paradigma de Depuração Depois do Teste.

2.5.1 Depuração baseada em Rastreamento e Inspeção

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 (Adams e Muchnick, 1986; Stallman e Pesch, 1999), 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 e 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; Balzer, 1969; Lieberman e Fry, 1998; Tolmach e 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) (Austin et al., 1994; Hastings e Joyce, 1992). As ferramentas FORMAN (Auguston, 1995) e CCI (Templer e Jefery, 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 e 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 e Jefery, 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 paradigma 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.

2.5.2 Depuração com Asserções

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 (2000, 2001), 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 Figura 2.7, 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 Figura 2.8 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 paradigma 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.

2.5.3 *Fatiamento* de Programas

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 e Horgan (1990) e Korel e 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

```
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;

@*/
}
```

Figura 2.7: 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); }
```

Figura 2.8: Rotina ativada pela violação de uma asserção (Rosenblum, 1995).

em língua portuguesa. A seguir é discutida a aplicação do *fatiamento* de programas na depuração.

Fatiamento Estático e Dinâmico

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 paradigma 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 e 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 e 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).

Fatiamento de Programas usando Informação de Teste

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*. A análise de mutantes visa à seleção de casos de teste que sejam capazes de identificar a presença dos desvios sintáticos (defeitos) mais comuns. Para determinar um conjunto de casos de teste com esta propriedade, programas mutantes são gerados pela aplicação de *operadores de mutação*. Os operadores são regras que são aplicadas para definir alterações no programa em teste. Por exemplo, o operador *elminação 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. 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. (1995, 1998) 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 paradigma 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 baseadas em *Fatias*

Heurísticas utilizando *fatias* têm sido propostas para reduzir o espaço de busca para localização do defeito (Agrawal, 1991; Agrawal et al., 1995; Collofello e Cousins, 1987; Chen e Cheung, 1997;

Lyle e Weiser, 1987; Pan e Spafford, 1992; 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 e Spafford, 1992; Pan, 1993).

Lyle e 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* (Agrawal, 1991; Chen e Cheung, 1997; Pan e Spafford, 1992; 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 e Spafford, 1992; Pan, 1993). Esta idéia foi inicialmente proposta por Collofello e 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 e 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 paradigma 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.

2.5.4 Depuração Algorítmica

Depuração Algorítmica Inter-Procedimental

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 e 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 paradigma 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.

Depuração Algorítmica Intra-Procedimental

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, 1993; Shimomura et al., 1995) 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^p e não há nenhuma definição

No.	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);`

Figura 2.9: 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) }

Figura 2.10: Execução do programa da Figura 2.9 para o caso de teste $\mathtt{n} = 2$ e $\mathtt{a} = (6,2)$.

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 Figura 2.9 para ilustrar o algoritmo de localização implementado em PELAS. As entradas $\mathtt{n} = 2$ e $\mathtt{a} = (6,2)$ produzem a saída incorreta $\mathtt{s} = 4$ (a saída correta é $\mathtt{s} = 8$). A Figura 2.10 contém os pontos de execução do programa para este caso de teste; e a Figura 2.11 contém a rede de dependência.

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

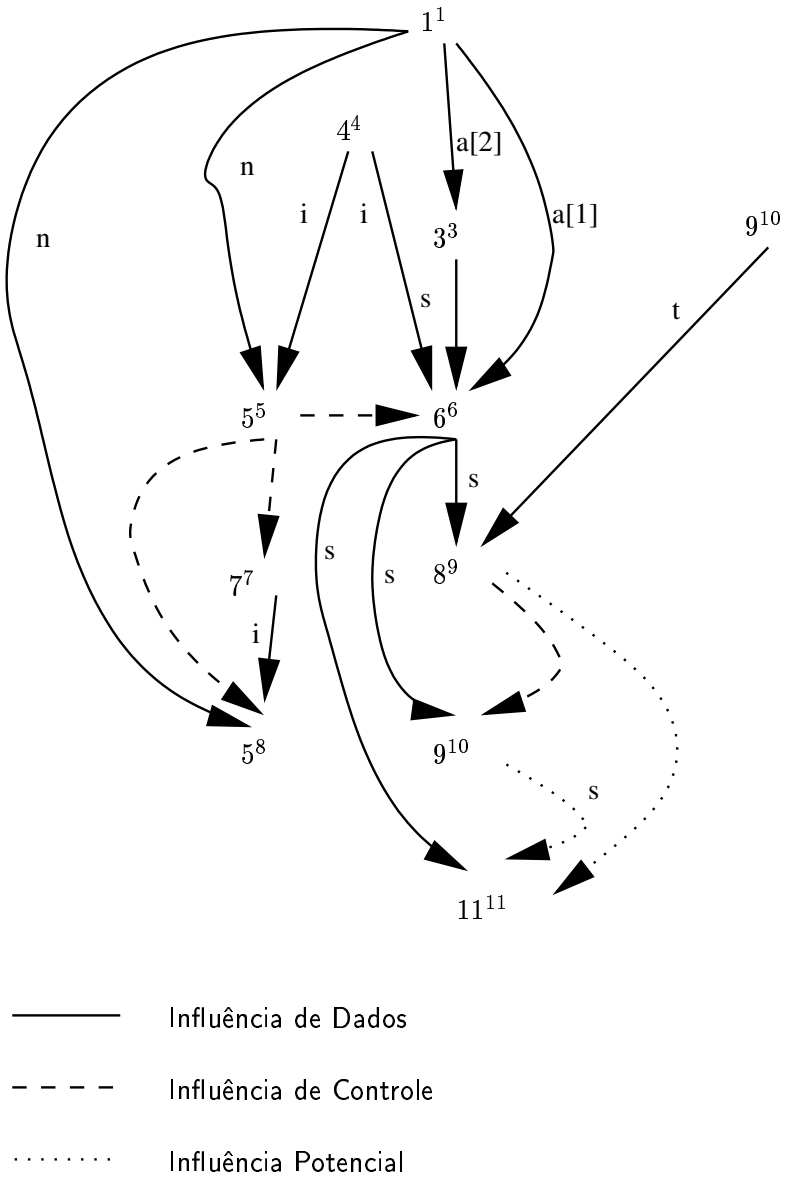


Figura 2.11: Rede de dependência gerada por PELAS durante a execução do programa.

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 e 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 (Korel e Yalamanchili, 1994; Gyimóthy et al., 1999). 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 (Nishimatsu et al., 1999; Wong 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 paradigma DDT; porém, utilizando informação básica de teste.

2.5.5 Discussão

A Tabela 2.1 apresenta os passos do paradigma 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 paradigma DDT. Isto é esperado visto que as técnicas dão ênfase a diferentes problemas colocados pelos passos do paradigma.

A técnica de depuração baseada em rastreamento e inspeção é fundamental ao paradigma 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 paradigma 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

Tabela 2.1: Adequação das técnicas de depuração ao paradigma DDT

Técnica de Depuração	Passo do Paradigma 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	

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 e Rilling, 1997, 1998) e heurísticas (Chen e Cheung, 1997; Pan e Spafford, 1992) 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 paradigma 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 paradigma DDT visto que são determinadas utilizando resultados coletados durante o teste e o paradigma 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 paradigma 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 paradigma DDT.

2.6 Considerações Finais

Neste capítulo, a atividade de depuração que ocorre *depois* do teste de software foi analisada. As principais técnicas de depuração de programas procedimentais foram avaliadas tendo como guia o paradigma DDT e a sua escalabilidade para sistemas reais. O paradigma 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.

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 paradigma 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. Além disso, é necessário avaliar a *eficácia*, *eficiência* e o *custo* do uso da informação de teste na depuração por meio de experimentos.

Nos próximos capítulos, vários aspectos do uso de informação de teste estrutural na depuração são

investigados. Inicialmente, são analisadas a eficácia e o custo do uso de informação de teste de fluxo de dados mais precisa no teste de programas que utilizam ponteiros e campos de registros. Em seguida, o uso de requisitos de teste estrutural é investigado quanto a sua habilidade na localização de defeitos por meio de heurísticas. A partir dos resultados dessa investigação, é proposta uma estratégia de depuração na qual a informação de teste é utilizada como *guia* de todo o processo de localização de defeitos. Esta estratégia é baseada em mecanismos que têm como objetivo utilizar a informação coletada durante o teste estrutural na *avaliação de possíveis sintomas internos* (passo 3) e na *seleção de novos possíveis sintomas internos* (passo 4).

Capítulo 3

Teste de Fluxo de Dados de Programas com Ponteiros e Registros

Neste capítulo, são apresentados dois modelos mais precisos de análise de fluxo de dados voltados para o teste de programas que utilizam ponteiros e campos de registros. Os modelos propostos são baseados em uma abordagem conservadora e foram implementados na ferramenta POKE-TOOL. A conjectura é que a análise de fluxo de dados mais precisa aumenta a eficácia do teste a um custo razoável. Para investigar esta conjectura, um estudo de caso foi realizado para avaliar a eficácia e o custo do teste utilizando os dois modelos mais precisos.

Do ponto de vista da *depuração depois do teste*, a análise de fluxo de dados mais precisa fornece informação mais detalhada para apoiar a depuração. Entretanto, ela somente será útil se o seu custo de obtenção for razoável, de forma a permitir que as técnicas de depuração baseadas em informação de teste sejam escaláveis.

3.1 Motivação

As definições dos critérios de teste baseados em fluxo de dados são abstratas e não estabelecem como os conceitos de fluxos de dados básicos (e.g., *definição*, *uso* e *associação definição-(potencial)-uso*) são implementados para uma linguagem em particular. Porém, a aplicação desses critérios em programas reais requer o uso de um *modelo de dados* que define como esses conceitos são tratados com respeito aos recursos mais comuns das linguagens de programação, tais como, variáveis agregadas (vetores e matrizes), variáveis estruturadas (registros) e variáveis derreferenciadas (endereços de memória obtidos pela derreferenciação de ponteiros) (Ghezzi e Jazayery, 1987). O desenvolvedor de ferramentas deve, portanto, selecionar um ou mais modelos de dados para apoiar o teste na linguagem alvo da sua ferramenta. Essa decisão tem implicações importantes não somente no custo e na eficácia dos critérios de fluxo de dados (como será discutido a seguir), mas também na posição relativa desses critérios na hierarquia de critérios de teste estruturais (Frankl e Weyuker, 1988; Horgan e London, 1991; Vergilio,

1997).

As primeiras versões das ferramentas de teste de fluxo de dados não consideravam os fluxos que envolviam a derreferenciação de ponteiros e o uso de campos de registros (Chaim, 1991; Frankl et al., 1985; Silva, 1995). No entanto, os defeitos devido ao uso incorreto desses recursos são comuns. Duncan e Robson (1996) desenvolveram um estudo exploratório dos defeitos mais comuns em programas escritos em linguagem C no qual mais de 70% dos programadores entrevistados responderam que os defeitos relativos ao uso incorreto de ponteiros estavam entre os mais freqüentes. Portanto, abordagens de teste de fluxo de dados que tenham como alvo a detecção e localização desses defeitos são necessárias e desejáveis.

Este problema tem sido abordado na literatura aumentando-se a precisão da análise de fluxo de dados (Horgan e London, 1991; Marx e Frankl, 1996, 1999; Ostrand e Weyuker, 1991; Vilela et al., 1997). A conjectura é que a análise de fluxo de dados mais precisa aumenta a eficácia do teste a um custo razoável (Marx e Frankl, 1996, 1999). O nível de precisão, por sua vez, é definido pelo modelo de dados implementado pela ferramenta de teste.

Algumas ferramentas utilizam modelos de dados que não distinguem as variáveis derreferenciadas das variáveis agregadas (Horgan e London, 1991). Essas ferramentas consideram todos os endereços de memória possíveis de serem associados a uma variável derreferenciada como um único *objeto de memória*. Como conseqüência, uma atribuição (ou referência) a qualquer elemento do objeto de memória significa uma definição (uso) de todo objeto. A vantagem dessa abordagem é a sua simplicidade visto que requer apenas o registro do caminho executado para a análise de adequação dos casos de teste. Por outro lado, ela pode produzir uma medida de cobertura *inflacionada*, pois não há garantia de que os fluxos de dados das associações *consideradas* como exercitadas tenham realmente ocorrido em tempo de execução ou de que aqueles que realmente ocorreram foram identificados por associações *exercitadas*.

Outra maneira de aumentar a precisão do modelo de dados é monitorando as posições de memória. Ostrand e Weyuker (1991) consideram associações de fluxo de dados de *posições de memória*, de tal forma que somente existe uma associação de fluxo de dados se a mesma posição de memória recebe um valor e este valor, inalterado, é subseqüentemente usado. A vantagem dessa abordagem é que ela captura *precisamente* os fluxos de dados estabelecidos nas *adu*s e *adpu*s. O efeito colateral é o custo adicional que ela acarreta devido à necessidade de monitoramento das posições de memória em tempo de execução.

Marx e Frankl (1996, 1999) estenderam o conceito de associação definição-uso para tratar ponteiros no teste de fluxo de dados. As associações definição-uso *estendidas* demandam a execução de caminhos livres de definição nos quais a posição de memória amarrada a uma variável derreferenciada não é alterada. De maneira semelhante à abordagem dinâmica de Ostrand e Weyuker (1991), a abordagem rastreia *precisamente* as cadeias definição-uso das variáveis derreferenciadas, porém sem monitorar as posições de memória. Entretanto, esta solução tem implicações em termos de custo visto que o algoritmo para determinar as associações estendidas possui complexidade assintótica *super-exponencial* e pode

requerer associações adicionais (Marx e Frankl, 1996, 1999). Além disso, ele não é adequado para linguagens que permitem operações aritméticas com ponteiros, o que inviabiliza a sua utilização nas linguagens que mais usam esse recurso (e.g., C, C++).

Neste capítulo, são apresentados dois modelos mais precisos de análise de fluxo de dados voltados para o teste de programas que utilizam ponteiros e campos de registros. Os modelos propostos, chamados de **nível 1** e **nível 2**, implementam níveis incrementalmente maiores de precisão na análise de fluxo de dados e são baseados na abordagem conservadora (Maldonado, 1991; Vilela et al., 1997). Esta abordagem trata as posições de memória que *podem* ser endereçadas pelas variáveis derreferenciadas como um objeto comum de memória; porém, quando comparada com outras abordagens, ela é mais conservadora na determinação dos requisitos de teste.

Para verificar a conjectura de que a eficácia aumenta a um custo razoável, foi realizado um estudo de caso. Nesse estudo, a eficácia e o custo do teste utilizando os modelos **nível 1** e **nível 2** foram comparados aos do teste utilizando um modelo (**nível 0**) que não leva em consideração as definições e usos relativos à derreferenciação de ponteiros ou aos campos de registros. Os três modelos foram implementados na ferramenta POKE-TOOL e a *eficácia* e o *custo* foram analisados com respeito a várias versões *defeituosas* do programa `Sort`.

Do ponto de vista da depuração baseada em informação de teste estrutural, a escolha do modelo de dados tem implicações importantes. Por um lado, é desejável que defeitos comuns e difíceis de detectar sejam devidamente tratados durante o teste para que o processo de depuração seja ativado caso eles estejam presentes. Além disso, a análise de fluxo de dados mais precisa fornece informação adicional útil para a localização de defeitos desse tipo. Por outro lado, as técnicas de depuração desenvolvidas nos capítulos seguintes são baseadas em informações coletadas durante o teste de fluxo de dados e essas técnicas somente são escaláveis se o custo do teste puder ser amortizado. Daí a necessidade de modelos de dados *eficazes* e *econômicos*.

O restante deste capítulo é organizado da seguinte forma. Na Seção 3.2, os modelos de dados baseados na abordagem conservadora são descritos. O estudo de caso e os resultados obtidos são apresentados na Seção 3.3. A Seção 3.4 contém uma análise dos resultados e a Seção 3.5 as considerações finais. A notação e o jargão utilizados são relativos à linguagem C.

3.2 Modelos de Dados baseados na Abordagem Conservadora

3.2.1 A Abordagem Conservadora

Vilela et al. (1997) propuseram a utilização da *abordagem conservadora* no teste de fluxo de dados de programas com ponteiros e campos de registros. Esta abordagem foi inicialmente definida por Maldonado (1991) para tratar variáveis agregadas e parâmetros passados por referência na ferramenta POKE-TOOL. Ela parte do princípio de que é razoável superestimar o número de requisitos de teste

para aumentar a sua eficácia.

Na análise dos fluxos de dados relativos a ponteiros e campos de registos, a abordagem conservadora é similar à de Horgan e London (1991) no sentido de que as variáveis derreferenciadas são tratadas da mesma forma que as variáveis agregadas. Porém, ela requer um número maior de requisitos de teste devido ao seu carácter conservador. Assim, segundo essa abordagem, qualquer atribuição (ou referência) a uma posição de memória endereçada utilizando expressões envolvendo ponteiros (e.g., `*(p+1)` onde `p` é um ponteiro) aponta para o mesmo objeto comum de memória (e.g., `*p`). As definições de variáveis derreferenciadas dão origem a associações de fluxo de dados da mesma maneira que as definições de variáveis escalares, porém, diferentemente dessas últimas, não bloqueiam definições anteriores, o que implica um aumento no número de requisitos de teste exigidos pelo critério de flluxo de dados, como exemplificado a seguir.

Considere os comandos 34, 40 e 50 do programa exemplo 2 da Figura 3.1. De acordo com a abordagem de Horgan e London (1991), não há distinção entre variáveis escalares, agregadas e derreferenciadas. Dessa maneira, a definição de `*p` em 34 não gera associações de fluxo de dados porque a definição de `*p` no comando 40 bloqueia a definição prévia no comando 34. Já de acordo com a abordagem conservadora a definição de `*p` no comando 40 não bloqueia a definição prévia em 34, resultando na exigência das associações (34,50,`*p`) e (40,50,`*p`), uma para cada atribuição a `*p`. A idéia é que as atribuições a uma variável derreferenciada não devem bloquear a exigência de outras associações de fluxo de dados (com respeito à mesma variável) porque em geral não é possível saber em tempo de compilação qual posição de memória está sendo alterada.

A desvantagem da abordagem conservadora são as associações adicionais que ela requer devido aos seus requisitos de teste conservadores. Ela demanda mais associações do que a abordagem de Horgan e London (1991), o que implica um custo adicional para exercitá-las ou determinar sua *não-executabilidade*. Por outro lado, a abordagem conservadora minimiza a probabilidade de que fluxos de dados que ocorrem em tempo de execução deixem de ser exercitados pelo conjunto de casos de teste adequado visto que ela requer as associações (fluxos de dados) adicionais que *podem* ocorrer em tempo de execução. Outro aspecto importante é que ela não requer monitoramento de memória, exigindo tão somente o caminho executado pelo caso de teste para a análise de adequação.

3.2.2 Modelos de Dados para Tratamento de Ponteiros e Campos

O tratamento de ponteiros e campos de registros baseado na abordagem conservadora foi implementado na ferramenta POKE-TOOL. Três diferentes modelos de dados para tratamento de ponteiros e campos de registros estão disponíveis; eles são referenciados como **nível 0**, **nível 1** e **nível 2** porque fornecem níveis crescentes de precisão na análise de fluxo de dados.

```
        proc(condition1, condition2)
        int condition1, condition2;
        {
                int x, y, z, *p;
24:                z = 17;
25:                x = 13;
26:                if(condition1) {
28:                        p=&y;
29:                        *p=z;
                }
                else {
33:                        p=&y;
34:                        *p=z;
35:                        switch(condition2) {
36:                              case 1:  p = &x;
37:                                       break;

38:                              case 2:  p = &z;
39:                                       break;
                         }
40:                        *p = 7 + z;
42:                        y = 53;
                }
49:                x = x + y + z;
50:                *p = *p +5;
51:                y = x + y;

        }
```

Figura 3.1: Programa exemplo 2.

Nível 0

O **nível 0** não leva em consideração associações de fluxo de dados envolvendo derreferenciação de ponteiros ou campos de registros. Este nível implementa o modelo de dados original da POKE-TOOL e utiliza a abordagem conservadora original para o tratamento de vetores e passagem de parâmetros por referência. Assim, de acordo com o **nível 0**, um vetor é tratado como uma variável simples. Por exemplo, há uma definição e uso de um vetor `v` no comando `v[i] = 20+v[j];`, independentemente de qual elemento de `v` é realmente definido ou usado. Entretanto, a abordagem conservadora estabelece que as atribuições a elementos de vetores são *definições por referência* (*def-ref*). Este tipo de definição dá origem a novas associações, porém, os caminhos que incluem a *def-ref* continuam a ser livres de definição com relação a outras definições da mesma variável, ou seja, uma *def-ref* não bloqueia a geração de associações de fluxo de dados devido a definições prévias. Com relação às chamadas de procedimentos neste modelo, sempre que uma passagem por referência ocorre (e.g., `foo(&v)`), há uma *def-ref* e um uso do parâmetro real (e.g., `v`).

Nível 1

O **nível 1** implementa a abordagem conservadora para tratar os fluxos de dados relativos a derreferenciação de ponteiros e a campos de registros. De acordo com esse modelo, uma *variável derreferenciada* `*p` é definida por uma operação de derreferenciação sobre uma expressão envolvendo o ponteiro `p`. Por exemplo, as variáveis derreferenciadas `*p` e `**q` ocorrem, respectivamente, em expressões `*(p+i)` e `*q[i+4]`. Outra diferença entre os modelos **nível 1** e **nível 0** é que cada campo de uma variável estruturada (registro) é considerado como uma variável independente. Assim, se `r` é uma variável do tipo registro, cada campo `r.field` é uma variável diferente. De igual maneira, se a variável derreferenciada é do tipo registro (e.g., `*p`), então cada campo (e.g., `*p.field`) representa uma variável derreferenciada diferente. As variáveis derreferenciadas são tratadas da mesma forma que os vetores na abordagem conservadora. Portanto, uma atribuição a uma variável derreferenciada implica uma *def-ref*. Qualquer outra ocorrência é um uso.

Diferentemente de Horgan e London (1991), na abordagem conservadora, os resultados da aplicação do operador de derreferenciação uma vez (e.g., `*p`) ou duas vezes (e.g., `**p`) sobre um ponteiro (e.g., `p`) resultam em variáveis derreferenciadas distintas. A razão para esta distinção deve-se ao fato de que o tipo dos elementos `*p` é diferente do tipo dos elementos `**p`; logo, as variáveis derreferenciadas apontam para objetos de memória distintos. Além disso, a abordagem conservadora foi estendida para tratar as situações em que ponteiros e variáveis estruturadas (registros) são utilizados como parâmetros em chamadas de funções. Considere-se a função a seguir em que `v` é um ponteiro para um tipo `T`. O modelo de de dados **nível 1** identifica as seguintes definições e usos:

```
/* 14 */                if((c = (*compare)(*hp, *j)) == 0)
/* 15 */                {
/* 15 */                   ++hp;
/* 15 */                   t= * hp ;
/* 15 */                   * hp = *  j ;
/* 15 */                   *  j =t ;
/* 15 */                   goto loop;
/* 15 */                }
```

Figura 3.2: Um pedaço de código do programa `Sort`.

```
foo(v, ...)
```

1. *c-uso* do ponteiro `v`;

2. *def-ref* de `*v`;

3. possível *c-uso* de `*v` (como foi considerada a existência de uma *def-ref* de `*v`, por uma questão de uniformidade, um *c-uso* de `*v` deve também ser considerado na chamada de procedimento);

4. Se `T` é um registro, então todo campo de `*v` possui uma *def-ref* associada.

Se `v` é um ponteiro para ponteiro (e.g., `T ** v`), então o **nível 1** também estabelece a existência de uma *def-ref* de `**v` e um possível *c-uso* de `**v`.

Nível 2

O modelo de dados **nível 2** é uma extensão do **nível 1**. Ele trata as variáveis derreferenciadas e os campos de registros de maneira muito similar ao **nível 1**. A diferença está na forma como são tratadas as atribuições para os ponteiros. De acordo com o **nível 2**, quando um ponteiro `p` recebe um novo valor (i.e., é definido), a variável derreferenciada `*p` fica associada a uma nova posição de memória. Acontece que o valor de `*p` neste caso também é novo, o que implica uma redefinição de `*p` também. Estas definições *implícitas* de variáveis derreferenciadas não são levadas em consideração no **nível 1**.

A intuição subjacente ao **nível 2** é que toda nova atribuição a um ponteiro deve ativar o teste da variável derreferenciada associada, uma vez que o valor atribuído ao ponteiro pode estar errado e o teste da variável derreferenciada pode aumentar as chances de detecção de um possível defeito. Este raciocínio já foi utilizado antes por Marx e Frankl (Marx e Frankl, 1996, 1999) e Pande et al. (1994).

3.2.3 Exemplo de Utilização dos Modelos de Dados Mais Precisos: Programa `Sort`

A Figura 3.2 apresenta um pedaço do código do programa `Sort` (padrão do sistema Unix e cujo código fonte é apresentado no Apêndice A) onde os números entre comentários representam os nós do grafo de fluxo de controle associado. Considerando as variáveis definidas no nó 15, a Tabela 3.1 descreve o conjunto de *adu*s para o critério todos usos derivadas utilizando os **níveis 0**, **1** e **2**. A primeira linha da Tabela 3.1 indica que a *adu* (15,(13,14), `hp`) é requerida pelos três níveis de análise de fluxo de dados, enquanto que a *adu* (15,28, `**hp`), apresentada na última linha da Tabela 3.1, é requerida apenas pelo **nível 2**.

Pode-se observar na Tabela 3.1 que há um aumento de *adu*s requeridas que começam no nó 15 quando se vai do **nível 0** ao **nível 1**. As atribuições às variáveis derreferenciadas `*hp` e `*j` dão origem a associações **nível 1**, mas elas não são consideradas no **nível 0**.

Com relação ao **nível 2**, as associções adicionais são derivadas a partir de atribuições a ponteiros. Por exemplo, tanto `hp` e `j` são declaradas como `char **`. Como conseqüência, a atribuição a `hp` implica a definição tanto de `*hp` (que é explicitamente definida no nó) como de `**hp`. De maneira semelhante, `**j` é definida por causa da atribuição a `*j`. Essas definições não são tratadas no **nível 1**.

A Figura 3.3 descreve outro fragmento de código do programa `Sort` no qual variáveis do tipo ponteiro recebem um novo valor. A Tabela 3.2 compara o conjunto de associações derivadas a partir do nó 4 e que envolvem o ramo (75,77) para o critério todos potenciais usos de acordo com os três modelos de dados. Ela está organizada em três seções nas quais as *adpu*s são derivadas utilizando diferentes modelos de dados. Por exemplo, a *adpu* (4,(75,77), { `la, lb` }) foi derivada utilizando **nível 0**, enquanto as *adpu*s (4,(75,77),{`la, lb`}) e (4,(75,77),{`*pa, *pb, fp->code, *(fp->code), fp->ignore, *(fp->ignore), fp->nflg, fp->rflg, fp->bflg, fp->m, fp -> n`}) foram derivadas utilizando o **nível 1**.

Note-se que várias variáveis definidas no nó 4 são incluídas na *adpu* e que o ramo essencial estendido deve ser atingido por um caminho livre de definição c.r.a. todas essas variáveis. O número de variáveis definidas no nó 4 (Figura 3.3) varia dependendo do modelo de dados. Utilizando o **nível 0**, apenas as variáveis que recebem novos valores via operador de atribuição são definidas (`la`, `lb`, `pa` e `pb`) visto que há apenas uso dos parâmetros reais nas chamadas de função de acordo com o **nível 0**. Por outro lado, utilizando o **nível 1**, os parâmetros reais na chamada de função são definidos por referência (*def-ref*); por isso, as variáveis `*la, *lb, *pa, *pb, fp->code, *(fp->code), fp->ignore, *(fp->ignore), fp->nflg, fp->rflg, fp->bflg, fp->m` e `fp -> n` são consideradas definidas.

As diferenças entre os **níveis 1** e **2**, no entanto, são sutis. A primeira diferença diz respeito às definições implícitas de variáveis. No **nível 1**, as variáveis `*pa` e `*pb` (impressas em itálico) estão *vivas* (uma variável v está *viva* em um nó j se existe uma definição de v em um nó i que atinge j por um caminho livre de definição) ao atingir o ramo (75,77), o que não ocorre quando o **nível 2** é utilizado.

Tabela 3.1: Associações requeridas pelo critério todos usos para os modelos **nível 0**, **nível 1** e **nível 2**

No.	Associações **nível 0**	Associações **nível 1**	Associações **nível 2**
1	(15,(13,14), hp)	(15,(13,14), hp)	(15,(13,14), hp)
2	(15,(13,21), hp)	(15,(13,21), hp	(15,(13,21), hp
3	(15,14, hp)	(15,14, hp)	(15,14, hp)
4	(15,15, hp)	(15,15, hp)	(15,15, hp)
5	(15,18, hp)	(15,18, hp)	(15,18, hp)
6	(15,(24,25), hp)	(15,(24,25), hp)	(15,(24,25), hp)
7	(15,(24,26), hp)	(15,(24,26), hp)	(15,(24,26), hp)
8	(15,(27,28), hp)	(15,(27,28), hp)	(15,(27,28), hp)
9	(15,(27,29), hp)	(15,(27,29), hp)	(15,(27,29), hp)
10	(15,28, hp)	(15,28, hp)	(15,28, hp)
11	(15,29, hp)	(15,29, hp)	(15,29, hp)
12	(15,31, hp)	(15,31, hp)	(15,31, hp)
13	(15,15, t)	(15,15, t)	(15,15, t)
14	(15,18, t)	(15,18, t)	(15,18, t)
15	(15,19, t)	(15,19, t)	(15,19, t)
16	(15,8, t)	(15,8, t)	(15,8, t)
17	(15,31, t)	(15,31, t)	(15,31, t)
18	—	(15,14, *j)	(15,14, *j)
19	—	(15,15, *j)	(15,15, *j)
20	—	(15,18, *j)	(15,18, *j)
21	—	(15,19, *j)	(15,19, *j)
22	—	(15,31, *j)	(15,31, *j)
23	—	(15,14, *hp)	(15,14, *hp)
24	—	(15,15, *hp)	(15,15, *hp)
25	—	(15,18, *hp)	(15,18, *hp)
26	—	(15,28, *hp)	(15,28, *hp)
27	—	—	(15,14, **j)
28	—	—	(15,14, **hp)
29	—	—	(15,28, **hp)

```
/*  1  2 78 */     for(k = nfields>0;k<=nfields;k++)
/*  3 */           {
/*  3 */              fp = &fields[k];
/*  3 */              pa = i;
/*  3 */              pb = j;
/*  3 */              if(k)
/*  4 */              {
/*  4 */                 la = skip(pa, fp, 1);
/*  4 */                 pa = skip(pa, fp, 0);
/*  4 */                 lb = skip(pb, fp, 1);
/*  4 */                 pb = skip(pb, fp, 0);
/*  4 */              }
/*  5 */              else
/*  5 */              {
/*  5 */                 la = eol(pa);
/*  5 */                 lb = eol(pb);
/*  5 */              }
...
/* 75 */              if((sa = code[*pb++]-code[*pa++]) == 0)
/* 76 */                 goto loop;
/* 77 */              return(sa*fp->rflg);
...
/* 78 */           }
```

Figura 3.3: Outro fragmento de código do programa `Sort`.

Tabela 3.2: Associações requeridas pelo critério todos potenciais usos envolvendo o nó 4 e o ramo (75,77) para os três modelos de dados

No.	Associações **nível 0**
1	(4,(75,77),{ la, lb})

No.	Associações **nível 1**
2	(4,(75,77),{la, lb})
3	(4,(75,77),{*pa, *pb, fp->code, *(fp->code), fp->ignore, *(fp->ignore), fp->nflg, fp->rflg, fp->bflg, fp->m, fp->n})

No.	Associações **nível 2**
4	(4,(75,77),{la, *la, lb, *lb, fp->code, *(fp->code), fp->ignore, *(fp->ignore), fp->nflg, fp->rflg, fp->bflg, fp->m, fp->n})

Isto acontece porque em qualquer caminho do nó 4 até o ramo (75,77) os ponteiros `pa` e `pb` foram redefinidos; porém, no **nível 1**, as atribuições implícitas a variáveis derreferenciadas não são capturadas, daí a razão porque `*pa` e `*pb` permanecem vivas. Por outro lado, no **nível 2**, as variáveis `*la` e `*lb` (também impressas em itálico) são implicitamente definidas quando os ponteiros `la` e `lb` recebem um valor no nó 4.

Outra diferença é que, no **nível 1**, os potenciais usos das variáveis `la` e `lb` estão separados em uma *adpu* diferente. Isto ocorre porque neste nível é possível atingir o ramo (75,77) a partir do nó 4 por caminhos livres de definição de dois tipos. Nos caminhos do primeiro tipo, as variáveis `la, lb, *pa, *pb, fp->code, *(fp->code), fp->ignore, *(fp->ignore), fp->nflg, fp->rflg, fp->bflg, fp->m` e `fp->n` estão vivas ao atingir o ramo (75,77); já nos caminhos do segundo tipo, todas essas variáveis estão vivas com exceção de `la` e `lb`. Esses caminhos envolvem duas ou mais iterações do laço sempre passando pelo nó 5 (onde ocorre redefinição de `la` e `lb`) para atingir o ramo (75,77); por isso, os potenciais usos relativos a `la` e `lb` são separados em uma *adpu* diferente (no Apêndice E é descrito em detalhes como são derivadas as *adpu*s para os critérios Potenciais Usos).

Observe-se, todavia, que quando o **nível 2** é utilizado só existem caminhos livre de definição do primeiro tipo. Isto acontece porque todos os caminhos do segundo tipo passam pelo nó 3 e todas as variáveis definidas no nó 4 são redefinidas ao ser atingido este nó. A diferença entre o **nível 1** e **nível 2** é que neste último as atribuições aos ponteiros `pa`, `pb` e `fp` no nó 3 implicam as redefinições implícitas de `*pa`, `*pb`, `fp->code`, `*(fp->code)`, `fp->ignore`, `*(fp->ignore)`, `fp->nflg`, `fp->rflg`, `fp->bflg`, `fp->m` e `fp->n`, o que não ocorre no **nível 1**.

A Tabela 3.3 descreve o número total de associações requeridas para o programa `Sort` para os critérios todos ramos, todos p-usos, todos usos, todos potenciais usos e todos potenciais usos/du para os diferentes níveis de análise fluxo de dados, bem como o aumento no número de associações provocado pela utilização dos níveis mais exigentes. Com relação a todos usos na Tabela 3.3, tem-se um total de 1678 associações requeridas utilizando **nível 0**, 2398 utilizando **nível 1** e 3238 utilizando **nível 2**. Do **nível 0** ao **nível 1**, 720 associações extras são requeridas, causando um aumento de 42,91% no número de associações requeridas (valor indicado entre parênteses); do **nível 1** para o **nível 2**, 840 novas associações são requeridas (aumento de 35,03%); e do **nível 0** para o **nível 2**, 1560 (aumento de 92,97%).

Tanto os critérios da família Fluxo de Dados como os da família Potenciais Usos são influenciados pelos modelos de dados. Entretanto, os critérios mais influenciados são todos p-usos e todos usos — por volta de 40% do **nível 0** ao **nível 1**; e 90% do **nível 0** para o **nível 2**. Para os critérios todos potenciais usos, o aumento é de 22,85% do **nível 0** para o **nível 1** e de 19,91% do **nível 0** para o **nível 2**.

A razão pela qual os critérios Potenciais Usos são menos impactados pelos diferentes modelos de dados é devido à maneira como são determinadas as *adpu*s usando ramos essenciais estendidos. Como elas envolvem várias variáveis definidas em um nó, o uso dos **níveis 1** e **2** exigirá uma *adpu* adicional

Tabela 3.3: Número de associações e o aumento causado pelos modelos de dados mais precisos

Critério	Modelo de Dados			Aumento entre os Modelos		
	0	1	2	0-1	1-2	0-2
Todos Ramos	249	249	249	0	0	0
Todos P-usos	1071	1478	2063	407 (38,00%)	585 (39,58%)	(992) 92.62%
Todos Usos	1678	2398	3238	720 (42,91%)	840 (35,03%)	1560 (92,97%)
Todos Potenciais Usos	4284	5263	5137	979 (22,85%)	-126 (-2,39%)	853 (19,91%)
Todos Potenciais Usos/Du	4284	5263	5137	979 (22,85%)	-126 (-2,39%)	853 (19,91%)

somente se uma definição de variável for detectada em um nó que não possuia uma variável definida de acordo com o **nível 0**.

Além disso, para os critérios Potenciais Usos, a introdução do **nível 2** levou a uma diminuição do número de associações em relação ao **nível 1**. Isto ocorre porque as definições implícitas de variáveis, que não são consideradas no **nível 1**, podem *bloquear* a existência de caminhos livres de definição que normalmente ocorrem com a utilização do **nível 1**. Esta situação ocorreu nas *adpu*s derivadas do trecho de código da Figura 3.3. Este fenômeno é menos comum nos critérios da família Fluxo de Dados porque cada *adu* possui apenas uma variável, o que provoca um aumento maior no número de requisitos de teste pela simples introdução de novas definições de variáveis; porém, ele pode também ocorrer, dependendo da distribuição das definições implícitas consideradas no **nível 2**.

De qualquer maneira, o número de requisitos de teste exigidos pelos critérios de teste de fluxo de dados aumenta ao passar do **nível 0** para um modelo de dados que leva em consideração ponteiros e campos de registros (**nível 1** ou **nível 2**). O impacto desse aumento vai depender do modelo de implementação utilizado pelos critérios escolhidos.

3.2.4 Análise de Custo

A seguir, os custos envolvidos na utilização de modelos de dados mais precisos são analisados. Inicialmente, discutem-se os custos envolvidos no teste de fluxo de dados:

Geração de Casos de Teste (C_1): o testador deve desenvolver um conjunto de casos de teste que seja adequado ao critério de fluxo de dados selecionado. Em geral, esse processo dá-se da seguinte maneira: um conjunto inicial de casos de teste é desenvolvido e a sua adequação ao critério é avaliada; se ele não é adequado ainda, então o conjunto deve ser aumentado com novos casos de teste até que se torne adequado.

Execução de Casos de Teste (C_2): os casos de teste devem ser executados para que sua validação possa ser realizada e as medidas de cobertura sejam coletadas.

Validação de Casos de Teste (C_3): as saídas dos casos de teste devem ser validadas em relação à especificação do programa para a determinação dos casos de teste que revelam defeitos.

Análise de Adequação (C_4): os dados coletados durante a execução dos casos de teste devem ser analisados para verificar a adequação do conjunto de casos de teste ao critério selecionado.

Análise de Não-executabilidade (C_5): este custo inclui a verificação da não-executabilidade das associações de fluxo de dados. Trata-se de uma atividade muito custosa visto que essencialmente exige intervenção humana. Uma maneira de evitar a intervenção humana é pela utilização de uma solução aproximada. Ela consiste na geração de uma massa de casos de teste grande de tal forma que se um requisito de teste não é executado pela massa é assumido como *não-executável*. Esta solução é também cara porque pode exigir muitas horas de processamento para a geração, execução e validação dos casos de teste da massa de teste.

A utilização dos níveis mais precisos de análise de fluxo de dados tem implicações importantes nos custos C_1, C_2, C_3, C_4 e C_5. O aumento no número de associações devido aos **níveis 1** e **2** tende a aumentar o custo C_1 visto que associações adicionais devem requerer casos de teste extras. Dessa maneira, espera-se um aumento no tamanho dos conjuntos de casos de teste do **nível 0** para os demais. O aumento no tamanho dos conjuntos de casos de teste tem um impacto direto em C_2, C_3 e C_4: conjuntos maiores exigem a execução, validação e análise de adequação de mais casos de teste.

O custo C_4, por sua vez, é definido por duas variáveis: *o número de nós* do caminho executado pelo caso de teste e o *número de associações*. O algoritmo de análise de adequação implementado em algumas ferramentas (e.g., ASSET, POKE-TOOL) é baseado em autômatos finitos de tal forma que cada *associação* é vinculada a um *autômato finito* (chamados de descritores). Basicamente, o algoritmo testa todos os autômatos para possíveis transições sempre que um novo nó é visitado. Uma associação é considerada exercitada se o seu autômato atinge o estado final. Portanto, o custo da análise de adequação é $O(c \times a)$ onde c é o número de nós do caminho executado e a o número de autômatos. Como o tamanho do caminho executado permanece constante, este custo pode ser reescrito como $O(a)$. Assim, o aumento do número de associações (número de autômatos) *per se* aumenta o custo C_4.

A análise de não-executabilidade envolve a identificação de padrões de não-executabilidade (caminhos do programa para os quais não existem dados de entrada que provoquem a sua execução) que são comparados com os possíveis caminhos que exercitariam uma *adu* ou *adpu*. A tarefa mais difícil é a identificação desses padrões. O uso de um modelo de dados mais exigente pode implicar a necessidade de identificação de novos padrões de não-executabilidade visto que as associações adicionais tendem a requer que novos caminhos sejam executados. Dessa maneira, C_5 deve aumentar com o crescimento do número de associações.

Da análise acima pode-se concluir que o *custo* de aplicação de um critério de fluxo de dados é principalmente determinado pelo *número de associações* e pelo *tamanho* do conjunto adequado de

Tabela 3.4: Defeitos relativos ao uso incorreto de ponteiros e campos de registros introduzidos no programa `Sort`

Defeito	Código Original	Código incorreto	ID
1	`if(b= *--ipb - *--ipa)`	`if(b= *--ipb - --*ipa)`	10,60
2	`while((*dp++ = *cp++) != '\n');`	`while((*++dp = *cp++) != '\n');`	13,77
3	`if(*--ipa != '0')`	`if(--*ipa != '0')`	11,53
4	`*pb == '\n' ? -fields[0].rflg`	`*pb == '\n' ? -fields[0].nflg`	23,23
5	`p->ignore = dict+128;`	`p->ignore = dict+290;`	7,17
6	`if (pb < lb && *pb==tabchar)`	`if (pb < lb || *pb==tabchar)`	4,63
7	`while(...ip[-1]->l<0)`	`while(...ip[-1]->l<=0)`	3,99

casos de teste. O aumento no número de *adu*s e *adpu*s causados pelo **nível 1** e **nível 2** certamente impacta os custos C_4 e C_5. Além disso, os modelos de dados que requerem mais associações tendem a gerar conjuntos de casos de teste maiores. Conjuntos maiores aumentam os custos C_1, C_2, C_3 e C_4. No experimento descrito a seguir, a *eficácia*, o *número de associações de fluxo de dados* e o *tamanho do conjunto de casos de teste* para diferentes critérios utilizando os três modelos de dados são analisados.

3.3 Estudo de Caso

3.3.1 Descrição do Experimento

Neste experimento, o programa `Sort`, padrão do ambiente Unix, foi utilizado. O `Sort` possui várias vantagens para o tipo de comparação realizada: ele faz uso exaustivo de ponteiros e campos de registros (todas as funções usam variáveis do tipo ponteiro); e, embora seja um programa pequeno, algumas das suas funções são bastante complexas.

Wong (1993) desenvolveu uma massa de teste de 997 entradas geradas aleatoriamente para o teste de 25 versões defeituosas do programa `Sort`. Delamaro et al. (2001a) analisaram a dificuldade de detecção dos defeitos dessas versões determinando o *índice de dificuldade* de cada um deles. Este índice foi determinado da segunte forma: 500 dados de teste foram gerados aleatoriamente e a porcentagem desses dados que revelavam o defeito foi calculada. Este processo foi repetido 30 vezes de forma que índice final de dificuldade foi definido pela média das 30 repetições. Das 11 versões defeituosas mais difíceis de detectar, foram identificadas sete versões que possuiam defeitos relacionados com o uso de ponteiros e campos de registros. A Tabela 3.4 descreve os sete defeitos selecionados bem como o índice de dificuldade (ID) de cada uma delas.

O procedimento experimental utilizado consistiu nos seguintes passos. Inicialmente, a análise estática de cada versão defeituosa do `Sort` foi realizada de acordo com os três modelos de dados. Em seguida, todos os 997 casos de teste pertencentes à massa de teste foram executados e a adequação de cada

caso de teste com respeito aos critérios selecionados — todos p-usos, todos usos, todos potenciais usos e todos potenciais usos/du — utilizando os três modelos de dados foi obtida.

As *adu*s e *adpu*s que não foram executadas por nenhum dos casos de teste da massa de teste foram consideradas *não-executáveis*. Portanto, os conjuntos de casos de teste considerados *adequados* com respeito a um critério são aqueles que possuem a maior medida de cobertura possível de se obter com a massa de teste.

O experimento consistiu na seleção de vários conjuntos de casos de teste adequados. Para cada um deles, foram determinados a *eficácia* — definida como o número de casos de teste reveladores de defeito no conjunto adequado — e o *tamanho* — definido como o número total de elementos do conjunto adequado. Utilizou-se o número de casos de teste reveladores de defeito como sendo a medida de eficácia porque todos os critérios de fluxo de dados, utilizando qualquer modelo, derivavam conjuntos adequados que incluíam pelo menos um caso de teste revelador de defeito. O tamanho dos conjuntos adequados, juntamente com o número de associações requeridas, estabelecem o *custo* de utilização de um critério de teste com um dado modelo de dados.

Os conjuntos adequados T foram obtidos da seguinte forma: casos de teste t da massa de teste eram escolhidos aleatoriamente e inseridos no conjunto T desde que pelo menos um requisito de teste diferente fosse executado; este processo era encerrado quando a máxima cobertura possível era atingida. Cem diferentes conjuntos de casos de teste adequados T foram selecionados e tiveram a sua eficácia e tamanho computados para cada versão defeituosa do `Sort` com respeito a cada critério de teste e modelo de dados.

A Tabela 3.5 apresenta a eficácia e o tamanho dos conjuntos adequados para os critérios todos ramos, todos p-usos, todos usos, todos potenciais usos e todos potenciais usos/du utilizando os modelos de dados **nível 0**, **nível 1** e **nível 2**. Por exemplo, a segunda linha refere-se aos dados relativos ao modelo de dados **nível 0** para o critério todos p-usos e as colunas indicam a versão defeituosa do `Sort`. Tanto as linhas como as colunas são subdivididas. As subdivisões das colunas indicam os dados relativos à eficácia (EF) e ao tamanho dos conjuntos adequados (Tam); as subdivisões das linhas referem-se à média (med) e ao desvio padrão (des) desses valores. Assim, para a versão 1, o critério todos p-usos no **nível 0** obteve os seguintes resultados: a eficácia média de 11,5 casos de teste reveladores de defeito com desvio padrão 2,1; o tamanho médio dos conjuntos adequados selecionados foi 53,2 casos de teste com desvio padrão 4,8. O critério todos ramos não é dividido em modelos de dados visto que se trata de um critério baseado unicamente em fluxo de controle.

A Tabela 3.6 mostra a eficácia de conjuntos de casos de teste de vários tamanhos selecionados aleatoriamente. Por exemplo, considere-se a linha que indica os conjuntos de casos de teste aletórios de tamanho 50; eles apresentam para o defeito 1 uma média de 5,98 de casos de teste reveladores de defeito e um desvio padrão de 2,29 para cem repetições.

Os resultados do experimento são também apresentados na forma de gráficos. Nas Figuras 3.4, 3.5, 3.6 e 3.7 são apresentados gráficos que relacionam a eficácia e o tamanho dos conjuntos adequados

com as versões do programa `Sort`. Para observar o efeito do aumento do número de casos de teste na eficácia, os dados relativos a conjuntos de casos de teste gerados aleatoriamente foram incluídos nos gráficos. O tamanho desses conjuntos aleatórios compreendem a faixa de variação do tamanho dos conjuntos adequados utilizando os diferentes modelos de dados.

3.3.2 Resultados

A partir dos dados coletados, pôde-se observar que o comportamento dos critérios todos p-usos e todos usos, para os programas do experimento, não foi modificado significativamente quando os modelos de dados mais precisos foram utilizados. Note-se nos gráficos das Figuras 3.4 e 3.5 que os pontos relativos ao **níveis 0**, **1** e **2** praticamente se sobrepõem. O aumento da eficácia devido aos **níveis 1** e **2** foi sempre menor ou igual a 8% para todos p-usos e a 6% para todos usos. Além disso, para alguns defeitos, o **nível 0** também obteve desempenho ligeiramente melhor (7% para todos p-usos e 4% para todos usos). Com relação ao tamanho dos conjuntos de casos de teste, não houve aumento importante visto que este ficou restringido a 6% para todos p-usos e a 4% para todos usos.

Os critérios da família Potenciais Usos, porém, apresentaram maior sensibilidade aos modelos mais precisos de análise de fluxo de dados. Com relação à eficácia, considerando o critério todos potenciais usos, houve uma versão defeituosa (versão 4) na qual o **nível 2** obteve um desempenho marcadamente melhor que os **níveis 0** e **1** (eficácia 20% maior) e uma outra (versão 2) em que o desempenho foi razoavelmente melhor (eficácia 13% maior); para as demais versões defeituosas, os **níveis 1** e **2** obtiveram melhor eficácia do que o **nível 0**, mas não tão expressivamente melhor. O critério todos potenciais usos/du utilizando o **nível 2** também foi marcadamente melhor para as versões 4 (eficácia 23% maior) e 2 (eficácia 15% maior) enquanto que para as demais versões a eficácia não sofreu modificações marcantes.

O tamanho dos conjuntos adequados aos critérios Potenciais Usos aumentaram consistentemente quando foram utilizados modelos mais precisos como pode ser observado nos gráficos das Figuras 3.6b e 3.7b. Do **nível 0** ao **nível 2**, o aumento máximo do tamanho dos conjuntos adequados foi de 12% para os dois critérios Potenciais Usos.

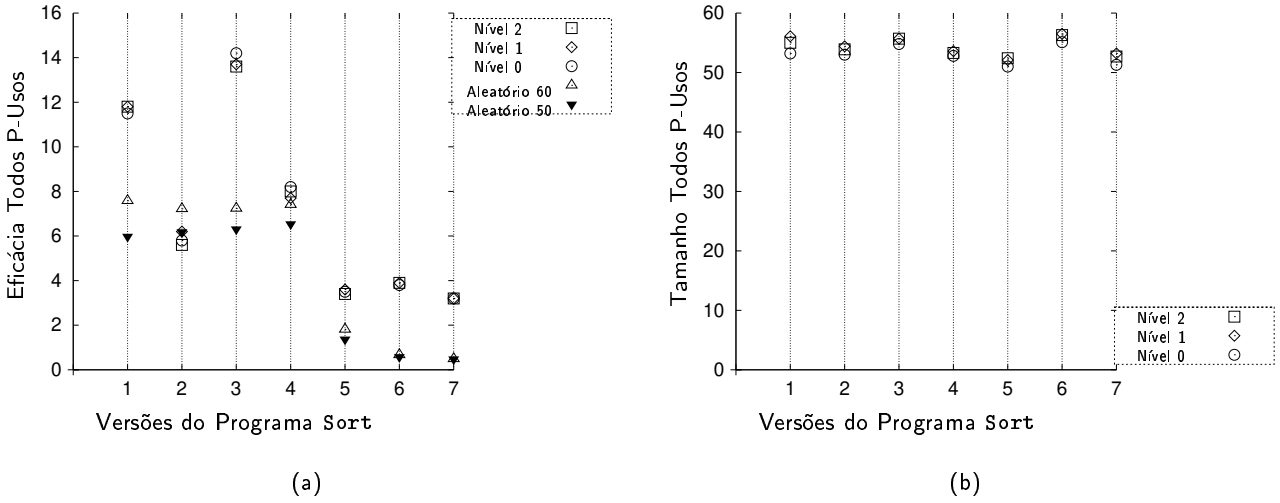
Os resultados dos conjuntos de casos de teste gerados aleatoriamente dão uma dimensão do efeito do aumento do número de casos de teste na eficácia. É interessante observar que nem sempre um conjunto de casos de teste maior implica um aumento marcante da eficácia. Isto pode ser observado nas versões 5, 6 e 7. Por outro lado, para algumas versões, um número maior de casos de teste tem um impacto direto na eficácia. Por exemplo, para a versão 2, o impacto na eficácia provocado pelo aumento no número de casos de teste é equivalente ou superior ao provocado pela introdução de modelos de dados mais precisos para os critérios Potenciais Usos. Além disso, para essa versão, o teste aleatório é mais eficaz que o teste de ramos e de fluxo de dados. O comportamento da versão 2 é discutido em mais detalhes na Subseção 3.4.

Tabela 3.5: Eficácia (EF) e tamanho (Tam) de conjuntos adequados a vários critérios de teste

Critério	Modelo de Dados		Defeitos 1 EF	1 Tam	2 EF	2 Tam	3 EF	3 Tam	4 EF	4 Tam	5 EF	5 Tam	6 EF	6 Tam	7 EF	7 Tam
Todos Ramos		med	6.6	31.3	3.4	31.0	6.4	30.0	4.0	30.7	1.3	31.1	1.6	30.6	1.4	30.2
		des	1.5	2.9	1.4	2.9	1.5	3.1	1.4	3.5	0.9	3.6	0.7	3.0	0.9	2.8
Todos P-usos	0	med	11.5	53.2	5.8	53.0	14.2	54.8	8.2	52.8	3.5	51.0	3.8	55.1	3.2	51.3
		des	2.1	4.8	1.6	5.0	2.5	5.9	2.2	5.1	1.1	5.7	1.1	5.2	1.1	5.0
	1	med	11.8	56.1	6.2	54.4	13.7	55.7	7.7	53.6	3.6	52.1	3.9	56.5	3.2	53.2
		des	2.3	5.2	1.9	5.5	2.3	4.3	2.0	4.6	1.2	4.4	1.0	5.2	1.2	4.8
	2	med	11.8	55.0	5.6	53.9	13.6	55.7	8.0	53.3	3.4	52.4	3.9	56.3	3.2	52.7
		des	2.3	5.2	2.2	5.2	2.6	5.0	2.1	4.5	1.0	5.3	1.0	4.9	1.2	5.3
Todos Usos	0	med	11.5	55.2	6.6	54.5	14.4	56.5	8.2	54.9	3.5	53.1	3.8	57.1	3.3	54.0
		des	2.4	5.8	1.8	5.0	2.5	5.0	2.2	5.3	1.1	4.9	1.1	4.9	1.1	5.1
	1	med	11.7	56.3	6.4	55.4	14.6	57.9	8.7	55.6	3.6	54.2	4.0	57.6	3.4	55.3
		des	2.4	4.8	2.3	5.7	2.3	5.5	2.2	5.3	1.2	5.4	1.1	5.1	1.1	5.3
	2	med	11.6	55.5	6.5	55.5	14.7	58.5	8.1	54.7	3.5	54.0	4.0	57.3	3.2	54.1
		des	1.9	4.1	1.8	6.1	2.6	5.4	2.3	5.0	1.2	5.5	1.0	5.6	1.0	4.9
Todos Potenciais Usos	0	med	17.4	74.2	7.7	69.9	18.0	73.0	10.7	70.3	4.7	70.3	4.0	74.0	3.6	71.4
		des	3.0	6.6	2.0	5.4	3.1	5.8	2.6	5.9	1.3	5.5	1.2	5.7	1.1	5.6
	1	med	17.3	78.4	8.3	75.0	18.3	77.7	11.4	76.3	4.8	75.2	4.0	79.8	3.7	76.9
		des	2.8	6.1	2.2	6.6	2.8	6.4	2.8	6.2	1.5	5.1	1.4	6.5	1.2	6.3
	2	med	18.2	81.0	8.7	78.4	19.4	79.4	12.8	77.1	4.7	76.5	3.9	80.6	3.7	78.4
		des	2.5	6.3	2.0	6.1	2.6	5.4	2.8	6.8	1.2	6.0	1.1	6.2	1.3	6.9
Todos Potenciais Usos/Du	0	med	17.3	78.1	7.9	74.7	18.5	77.5	11.6	76.1	4.9	75.7	4.0	79.1	5.2	75.0
		des	3.0	5.5	2.0	5.9	2.7	5.1	2.4	5.4	1.6	5.8	1.3	6.7	1.2	5.6
	1	med	18.1	84.9	8.6	82.3	19.1	85.2	12.2	83.0	4.8	82.3	4.0	85.6	5.8	82.6
		des	3.0	6.1	2.2	5.3	2.5	5.2	2.6	5.5	1.4	6.0	1.0	5.5	1.1	5.9
	2	med	18.6	87.0	9.0	84.1	19.7	84.9	14.2	83.8	4.8	83.8	4.1	86.5	5.8	83.4
		des	1.7	5.5	2.3	5.6	3.2	6.3	2.5	5.4	1.7	6.2	1.2	5.5	1.1	6.2

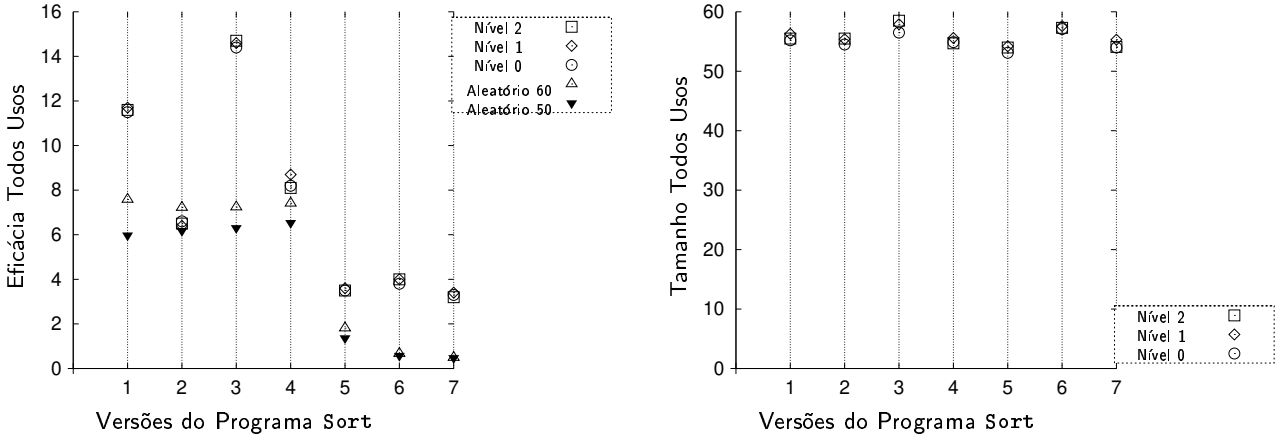
Tabela 3.6: Eficácia (EF) de conjuntos gerados aleatoriamente de diferentes tamanhos (Tam)

Tam		Defeitos 1	2	3	4	5	6	7
30	med	3.8	3.63	3.43	3.65	0.73	0.46	0.23
	des	1.74	1.7	1.5	1.64	0.72	0.74	0.47
40	med	4.8	4.77	4.9	5.14	0.99	0.48	0.31
	des	2.19	1.93	1.91	2.02	0.96	0.61	0.49
50	med	5.98	6.18	6.31	6.54	1.37	0.58	0.48
	des	2.29	2.06	2.37	2.38	1.12	0.71	0.66
55	med	6.03	6.46	6.67	6.88	1.41	0.65	0.41
	des	2.03	2.25	2.77	2.46	1	0.85	0.62
60	med	7.58	7.22	7.24	7.41	1.81	0.67	0.49
	des	2.43	2.35	2.43	2.31	1.29	0.77	0.76
65	med	7.63	8.02	7.76	8.1	1.58	0.96	0.58
	des	2.33	2.43	2.79	2.81	1.17	0.9	0.76
70	med	8.82	8.53	8.97	8.97	1.81	0.86	0.51
	des	2.76	2.61	2.87	2.63	1.4	0.82	0.69
75	med	8.94	9.08	9.24	9.51	2.02	1.11	0.64
	des	2.35	3.06	2.81	2.67	1.38	1.14	0.73
80	med	9.78	9.61	10.06	9.96	2.19	1.19	0.75
	des	3.11	2.96	3.14	2.76	1.32	0.98	0.73
85	med	10.41	10.54	10.2	10.57	2.18	0.99	0.74
	des	2.89	2.87	3.36	2.79	1.37	0.94	0.85
90	med	11	10.85	11.63	11.15	2.33	1.23	0.74
	des	2.76	3.25	2.66	3.06	1.46	1.09	0.82



(a) (b)

Figura 3.4: Eficácia e tamanho dos conjuntos adequados ao critério todos p-usos utilizando os diferentes modelos de dados.



(a) (b)

Figura 3.5: Eficácia e tamanho dos conjuntos adequados ao critério todos usos utilizando os diferentes modelos de dados.

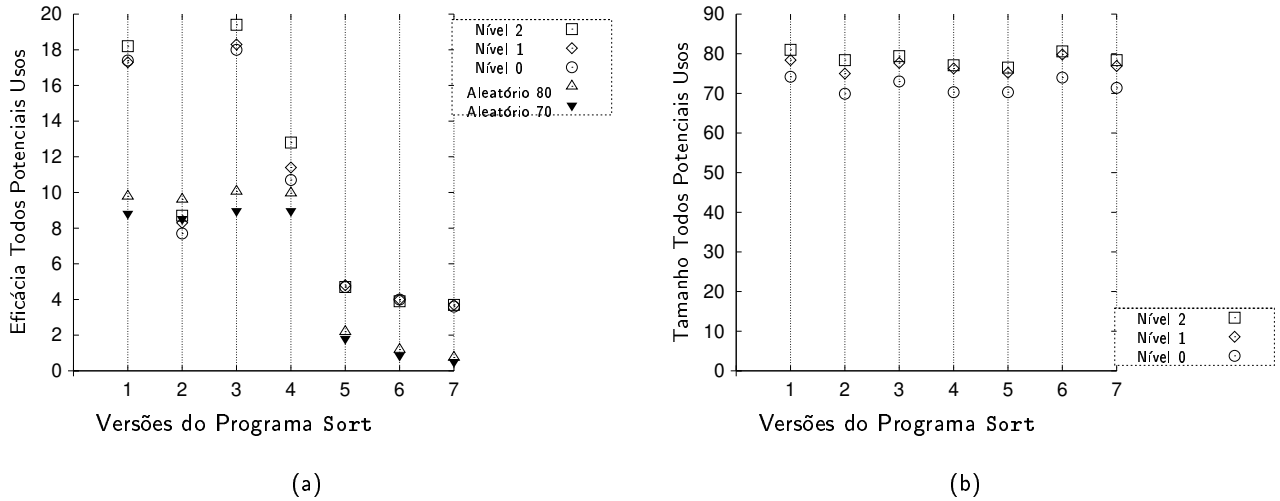


Figura 3.6: Eficácia e tamanho dos conjuntos adequados ao critério todos potenciais usos utilizando os diferentes modelos de dados.

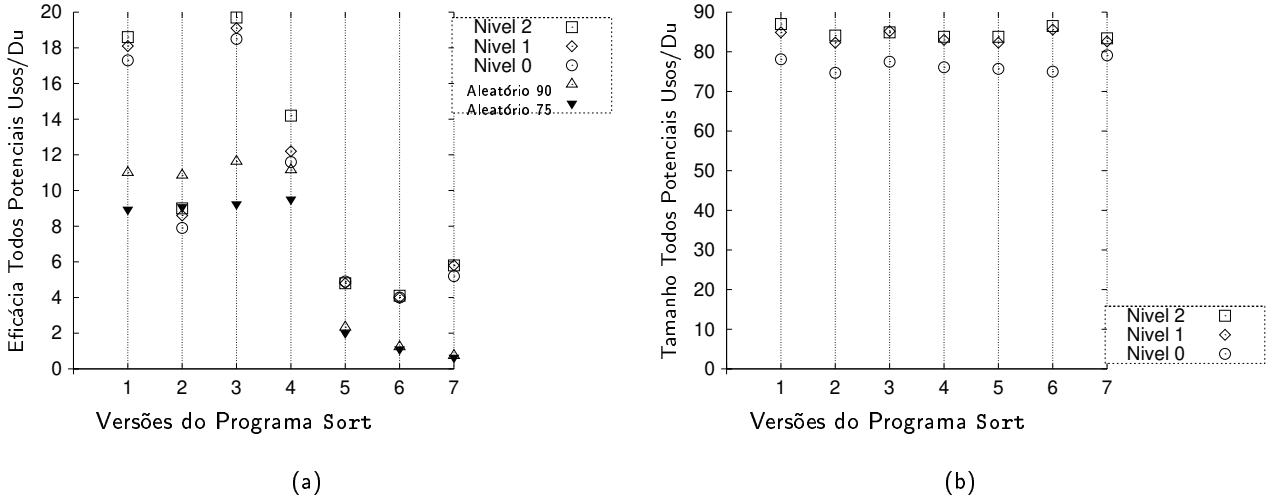


Figura 3.7: Eficácia e tamanho dos conjuntos adequados ao critério todos potenciais usos/du utilizando os diferentes modelos de dados.

3.3.3 Ameaças à Validade

Os resultados apresentados foram obtidos considerando-se algumas aproximações da realidade; por isso, eles devem ser analisados tendo em vista as limitações dessas aproximações. Uma delas é o fato de que se trata de um experimento baseado em um único programa, embora várias versões diferentes tenham sido utilizadas. Todavia, o programa `Sort` é complexo e utiliza extensivamente ponteiros, de forma que o que foi observado nele pode também ocorrer em outros programas de características semelhantes.

Outra limitação é que os defeitos foram semeados artificialmente e são únicos em cada versão. Na prática, observa-se que os programas em geral possuem vários defeitos. Deve-se ressaltar, entretanto, que o propósito do estudo de caso é verificar o desempenho dos diferentes modelos de dados na deteção de defeitos relacionados com ponteiros e campos de registros. Nesse sentido, o isolamento dos defeitos é interessante, pois permite rastrear o relacionamento entre o tipo de defeito e a eficácia do modelo de dados utilizado.

Os resultados devem ainda ser analisados levando em consideração as características da massa de teste e dos conjuntos adequados. A massa foi determinada aleatoriamente; porém, é razoável supor que em muitas situações os dados de entrada serão gerados por testadores, o que deve levar a dados de entrada com características distintas (e.g., concentrados em alguns pontos do domínio de entrada). Os conjuntos selecionados, por sua vez, são *aproximadamente* adequados, pois as associações não-executáveis não foram determinadas. Dessa maneira, as medidas de eficácia e tamanho dos conjuntos são subestimadas.

A última questão a ser considerada é o fato de os defeitos serem *fáceis* de detectar usando critérios de fluxo de dados. Essa limitação é superada utilizando o número de casos de teste reveladores de defeito como medida de eficácia. Essa medida visa dar uma indicação da habilidade de um modelo de dados em selecionar associações com maior probabilidade de detectar defeitos relativos ao uso incorreto de ponteiros e de campos de registros.

Portanto, é importante observar que os resultados do estudo de caso são influenciados por variáveis como o tipo do defeito, o domínio de aplicação e a estrutura do programa, bem como a arquitetura do sistema no qual ele está inserido.

3.4 Discussão

As versões defeituosas do `Sort` foram inspecionadas para determinar quais as razões do comportamento observado nos resultados. Para os critérios todos p-usos e todos usos, foi verificado que a menor sensibilidade à análise de fluxo de dados mais precisa é devida aos fluxos de controle e de dados particulares ao programa `Sort`. Três situações foram observadas.

1. Algumas associações adicionais requeridas pelos **níveis 1** e **2** foram *incluídas* (de acordo com a

definição de Marré e Bertolino (1996)) pelas associações de **nível 0**, o que significa dizer que qualquer caminho que exercita a associação **nível 0** também exercita uma ou mais associações adicionais **nível 1** ou **nível 2**. O caso típico envolve associações cujas variáveis compartilham um relacionamento baseado na linguagem de programação (e.g., ponteiro `p` e a variável derreferenciada `*p`). Muito freqüentemente no programa `Sort`, o caso de teste que satisfaz a associação relativa a `p` também satifaz a associação relativa a `*p`.

2. Outras associações de **nível 1** ou **nível 2** foram, por sua vez, *incluídas* pelas associações de **nível 0** devido a *condições de executabilidade*. O que ocorre neste caso é que aqueles caminhos que exercitariam as associações de **níveis 1** e **2**, mas não as de **nível 0**, são não-executáveis. Interessantemente, em geral, as variáveis nessas associações não compartilham relação baseada na linguagem de programação; ao contrário, as associações compartilham padrões de não-executabiliade contidos no código do `Sort` que terminam por determinar a relação de inclusão *dinâmica*.

3. Há ainda um grupo final de associações **níveis 1** e **2** que é *quase incluído* pelas associações **nível 0**. Essas associações não são incluídas nem estaticamente nem dinamicamente. Mesmo assim, elas possuem uma grande probabilidade de também serem incluídas, pois as condições que evitariam a inclusão são muito particulares. Por exemplo, uma associação adicional **nível 1** somente não seria incluída por outra associação **nível 0** se o laço que controla ambas as associações fosse executado uma única vez em todos os casos de teste.

Dessa maneira, como poucos novos caminhos além daqueles necessários para satisfazer as associações **nível 0** são exigidos, tanto a eficácia como o tamanho dos conjuntos de casos de teste adequados permaneceram praticamente constantes. Portanto, para as versões defeituosas do `Sort`, o custo dos componentes relacionados com o aumento no número de casos de teste são irrelevantes. Por outro lado, o custo associado com a análise de adequação (C_4) aumenta consideravelmente uma vez que o número de *adu*s pode aumentar em até 90% do **nível 0** para o **nível 2**.

Entretanto, o número de *adu*s pode ser reduzido visto que muitas das associações de **níveis 1** e **2** relativas às variáveis derreferenciadas são incluídas pelas associações de **nível 0** relativas aos ponteiros. Essas associações podem ser abstraídas em uma única associação de maneira semelhante às *adpu*s dos critérios Potenciais Usos. Uma abordagem mais geral, porém, seria determinar a relação de inclusão entre *adu*s utilizando o algoritmo proposto por Marré e Bertolino (1996).

Com relação aos critérios Potenciais Usos, observou-se que as novas *adpu*s exigidas pelos **níveis 1** e **2** são relevantes visto que elas são novos requisitos de teste que requerem novos casos de teste para serem satisfeitos. Este fato provoca aumento no tamanho dos conjuntos adequados de casos de teste. Como conseqüência, os custos influenciados pela cardinalidade dos conjuntos adequados de teste aumentam na mesma taxa. Além disso, o número maior de *adpu*s aumenta também os custos da análise

de adequação.

Diferentemente dos critérios todos p-usos e todos usos, a eficácia dos critérios da família Potenciais Usos para os programas do experimento mostrou-se sensível aos defeitos. Para a maioria deles (seis de sete), as associações devidas aos **níveis 1** e **2** não induziram à seleção de casos de teste adicionais que aumentassem marcadamente a eficácia. Entretanto, para a versão defeituosa 4, o aumento na eficácia foi de aproximadamente 20%, o que pode ser considerado significativo. A eficácia também aumentou na versão defeituosa 2 usando todos potenciais usos/du e o **nível 2**; porém, este aumento pode ser creditado ao aumento no número de casos de teste e não ao modelo de dados mais preciso.

A versão 4 do programa `Sort` foi inspecionada para determinar qual o papel do modelo de dados **nível 2** no aumento da eficácia. Para tanto, a *probabilidade* (referenciada como PRD) de cada *adpu* ser exercitada por um caso de teste *revelador de defeito* foi determinada. Como esperado, diferentes probabilidades foram encontradas para as *adpu*s de **níveis 0**, **1** e **2**. Várias associações de **nível 2** obtiveram valores de PRD altos; entretanto, há uma *adpu* desse nível em particular cuja PRD foi de 94%, enquanto que a *adpu* de **nível 0** que obteve melhor PRD atingiu 88%. É interessante observar, porém, que esta associação de **nível 2** ((31,(27,28), { `*i, **i` }) é exigida exclusivamente devido à análise de fluxo de dados mais precisa imposta pelo **nível 2** visto que ela não contém variável definida que seja escalar ou agregada. Dessa maneira, os caminhos que exercitam a associação e que possuem grande probabilidade de revelar o defeito foram induzidos pelo modelo de dados **nível 2**.

O desempenho dos critérios Potenciais Usos utilizando **nível 2** para a versão 2 é explicado quando são analisados os dados contidos na Tabela 3.6 e nas Figuras 3.6 e 3.7. Comparando o teste de ramos e de fluxo de dados com o teste aleatório (Tabela 3.6), observa-se que ambos obtiveram melhor desempenho do que o teste aleatório para os defeitos inseridos no `Sort`, com exceção do defeito da versão 2. Para esta versão, o aumento no tamanho dos conjuntos de casos de teste provoca uma melhora observável da eficácia que é equivalente ou superior à causada pelos uso de modelos de dados mais precisos nos critérios potenciais usos.

A razão para este comportamento é que as falhas nesta versão são observadas sempre que uma fusão (*merge*) de arquivos ocorre. Acontece que a chance de serem escolhidos dados de entrada que requeiram uma fusão de arquivos é maior utilizando teste aleatório do que utilizando critérios de teste estruturais. Isto porque o teste aleatório pode escolher vários dados de entrada de um mesmo subdomínio da entrada, o que em princípio não deve ocorrer nos critérios estruturais. Portanto, o aumento da eficácia pode não ter sido causado pelo uso do **nível 2**, mas pelo maior número de casos de teste. Já em relação à versão 4, o aumento da eficácia provocado pelo uso do **nível 2** supera aquele causado pelo simples aumento do número de casos de teste.

3.5 Considerações Finais

Neste capítulo foram apresentadas a definição e a implementação de dois modelos de dados mais precisos para apoiar o teste de programas que utilizam ponteiros e campos de registros. Esses dois novos modelos tiveram a sua *eficácia* e *custo* avaliados por meio de um estudo de caso no qual a base de comparação foi um modelo que não considera os fluxos de dados relativos à derreferenciação de ponteiros ou a campos de registros. No experimento realizado, sete versões *defeituosas* do programa `Sort` do ambiente Unix foram utilizadas. Este programa foi selecionado porque se trata de um programa complexo que utiliza extensivamente ponteiros e registros.

Durante o experimento, conjuntos de casos de teste adequados a quatro critérios de fluxo de dados (todos p-usos, todos usos, todos potenciais usos e todos potenciais usos/du) foram determinados utilizando os três modelos de dados. Para cada conjunto de casos de teste, foram calculados a eficácia e o tamanho. A eficácia é definida pelo número de casos de teste reveladores de defeito presentes no conjunto e o tamanho é definido pela sua cardinalidade. O custo de aplicação é determinado por duas variáveis: o número de associações requeridas e o tamanho dos conjuntos adequados.

Para os critérios da família Fluxo de Dados, todas as versões do program `Sort` apresentaram comportamento similar. Os modelos mais precisos de análise de fluxo de dados aumentaram o número de associações requeridas, mas a eficácia e o tamanho dos conjuntos adequados de casos de teste aumentaram marginalmente.

Esse comportamento é explicado pelas seguintes situações que ocorrem no programa `Sort`:

1. Uma parte das associações adicionais é *incluída* estaticamente pelas associações do modelo de dados menos preciso;

2. Algumas das associações adicionais são *dinamicamente incluídas* pelas associações do modelo menos preciso devido a condições de executabilidade; e ainda

3. Outras associações adicionais são *quase incluídas* pelas associações do modelo menos preciso, pois as condições para que não ocorra a inclusão são muito improváveis.

Esses três aspectos fizeram com que o aumento no tamanho dos conjuntos adequados fosse muito pequeno pois, de fato, poucos *novos* caminhos de teste foram requeridos. Como conseqüência, a eficácia também não teve um aumento marcante em decorrência da utilização dos modelos mais precisos de análise, embora o custo da análise de adequação tenha aumentado significativamente visto que o número de associações requeridas chega a ser até 90% maior.

Com relação aos critérios da família Potenciais Usos, a introdução de modelos de dados mais precisos aumentou o tamanho dos conjuntos de casos de teste adequados para as versões defeituosas do `Sort`, o que significa que as novas associações requeridas pelos modelos mais precisos exigem novos caminhos a serem testados. A eficácia, porém, aumentou marcadamente apenas para uma versão em decorrência

do uso dos modelos de dados mais precisos. Neste caso, o modelo mais preciso requer uma associação em particular que induz à seleção de casos de teste reveladores de defeito. O custo extra, por sua vez, é razoável uma vez que o número de casos de teste aumentou no máximo 12% e o número de associações 23%. Neste sentido, o custo adicional para os critérios Potenciais Usos parece ser compensador visto que o estudo de caso indica que o uso de modelos de dados mais precisos aumenta a probabilidade de os conjuntos adequados serem mais eficazes.

De acordo com Harrold (2000), o teste de *todos* os fluxos de dados relativos ao uso de ponteiros pode ser muito caro. Daí a necessidade de experimentos que avaliem a *eficácia* e o *custo* do teste de fluxo de dados de programas que utilizam esses recursos. O estudo de caso realizado indica que a eficácia na detecção de defeitos relativos ao uso incorreto de ponteiros e campos de registros depende do *programa* e do *defeito*. Para os critérios da família Fluxo de Dados, a estrutura do programa foi o fator determinante para os resultados observados, enquanto que as características dos defeitos também contribuíram para o desempenho dos critérios Potenciais Usos. Portanto, esta última observação indica que existem defeitos que podem ser mais facilmente detectados quando modelos de dados mais precisos são utilizados.

Do ponto de vista da *depuração depois do teste*, a variável mais importante é o custo de obtenção das informações mais detalhadas fornecidas pelos modelos de dados mais precisos. Neste sentido, o uso desses modelos para os critérios da família Fluxo de Dados tem um impacto significativo no custo da análise de adequação. Uma possível solução seria utilizar mecanismos de redução do número de associações requeridas, por exemplo, pela determinação da relação de inclusão estática entre as associações (Marré e Bertolino, 1996). Com relação aos critérios Potenciais Usos, o custo adicional é razoável porque o número de associações requeridas não cresce muito. Isto ocorre porque a forma como as *adpu*s são determinadas — incluindo várias variáveis e utilizando ramos essenciais — já reduz o número de associações requeridas.

Como última observação, é importante ressaltar que os resultados do estudo de caso apresentado são preliminares e novos experimentos são necessários para uma avaliação completa da eficácia e do custo da utilização de informação de fluxo de dados mais precisa na detecção de defeitos relativos ao uso incorreto de ponteiros e campos de registros.

Capítulo 4

Uso de Requisitos de Teste na Localização de Defeitos

Neste capítulo, é avaliado o uso dos requisitos de critérios de teste estruturais na depuração por meio de heurísticas. O objetivo é avaliar a habilidade de uma heurística H utilizando os requisitos de teste de um critério C em identificar informação útil para a localização do defeito. Essa informação pode ser um trecho de código onde está o defeito ou um subconjunto de requisitos que fornecem indicações úteis para a sua localização. Para capturar essas indicações em tempo de execução, é proposto o conceito de *requisito de teste revelador de erro*.

A avaliação realizada consistiu em um estudo de caso em que diferentes pares (H, C) foram aplicados em vários conjuntos de casos de teste (gerados segundo três cenários de teste-depuração) para localizar defeitos em versões incorretas do programa `Sort`. Os dados obtidos foram comparados considerando-se aspectos de *inclusão*, *eficiência* e *custo*.

Os pares (H, C) que obtiveram os melhores resultados foram mais *inclusivos* e *eficientes* na seleção de requisitos de teste reveladores de erros do que na seleção de um trecho de código que inclui o defeito. Este fato indica a necessidade de mecanismos que ajudem o mantenedor a identificar os requisitos reveladores de erro dentre aqueles selecionados pelos pares (H, C) e a selecionar novos requisitos a serem investigados, a partir dos requisitos reveladores de erro identificados, para guiá-lo até a localização precisa do defeito.

4.1 Motivação

A utilidade dos resultados do teste estrutural na depuração de software advém do fato de haver uma relação direta entre os requisitos de teste e a implementação do programa. Os comandos do trecho de código associado aos requisitos exercitados por um caso de teste revelador de defeito são candidatos naturais a estarem relacionados a um sintoma interno. Porém, conforme discutido no Capítulo 2, o número de comandos selecionados pode ainda ser grande. Daí a necessidade de mecanismos para reduzir

o número de requisitos de teste selecionados e, por conseguinte, o trecho de código a ser investigado. Esse número pode ser reduzido utilizando heurísticas.

Heurísticas que manipulam os mais variados artefatos (e.g., *fatias* de programas, requisitos de teste, caminhos executados) têm sido propostas baseadas na intuição de que há uma grande probabilidade do defeito estar contido no trecho de código identificado por elas (Agrawal, 1991; Agrawal et al., 1995; Collofello e Cousins, 1987; Chen e Cheung, 1997; Lyle e Weiser, 1987; Pan e Spafford, 1992; Pan, 1993). Entretanto, este objetivo — incluir o defeito — é muito restrito, pois não leva em consideração outras indicações, fornecidas em tempo de execução pelos requisitos de teste, que podem levar o mantenedor a localizar o defeito.

Para capturar essas indicações em tempo de execução, é definido o conceito de *requisito de teste revelador de erro*. Durante a execução de um caso de teste, um requisito pode ser exercitado uma ou mais vezes. Cada vez que o requisito é exercitado ocorre uma das suas *instâncias*. O requisito de teste é *revelador de erro* se o mantenedor, ao analisar as instâncias do requisito, responde negativamente a uma das seguintes perguntas:

- a instância do requisito de teste (nó, ramo, *adu*, *adpu*) deveria ter sido executada?
- o caminho que exercita a instância está correto?
- o valor atribuído e o valor (potencialmente) referenciado estão corretos (no caso de uma *adu* ou *adpu*)?
- o código executado pela instância está correto?

Em outras palavras, um requisito de teste é revelador de erro quando indica um comportamento incorreto ou um desvio da especificação causado pelo defeito, ou seja, um *erro* (Subseção 2.2).

Neste capítulo, é realizada uma comparação empírica do uso de requisitos de teste selecionados por meio de heurísticas na localização de defeitos. O objetivo é avaliar a habilidade de uma heurística H utilizando os requisitos de teste de um critério C em identificar requisitos de teste que contenham informação útil para a localização de um defeito. Esta informação útil pode ser um trecho de código onde está o defeito ou um subconjunto de requisitos de teste reveladores de erros.

A comparação realizada baseou-se em medidas de *inclusão*, *eficiência* e *custo*. A medida de *inclusão* tem como objetivo avaliar a capacidade do par (H, C) de selecionar requisitos de teste que contenham informação útil para a localização de um defeito. A *eficiência* mede o tamanho do espaço de busca identificado e o *custo* avalia a demanda computacional de se usar a heurística H e os requisitos de teste do critério C. O estudo empírico utilizou várias versões incorretas do programa `Sort`. Diferentes pares (H, C) foram aplicados e tiveram as suas medidas de inclusão, eficiência e custo coletadas para vários conjuntos de casos de teste em três cenários de teste-depuração.

A partir dos dados coletados, foi analisada a influência da heurística, do critério de teste utilizado e do cenário de teste-depuração na habilidade do par (H, C) em fornecer informação útil para a depuração.

Os resultados obtidos indicam que os pares (H, C) mais *inclusivos* e *eficientes* — portanto, os mais viáveis para aplicações reais — são melhores na identificação de requisitos reveladores de erro do que na seleção de um trecho de código que inclui o defeito. Este fato indica a necessidade de mecanismos que ajudem o mantenedor a identificar requisitos reveladores de erro dentre aqueles selecionados pelos pares (H, C) e a selecionar novos requisitos a serem investigados, a partir dos reveladores de erros identificados, para guiá-lo até a localização precisa do defeito.

A organização do restante do capítulo é a seguinte. Na próxima seção, são introduzidos os conceitos de instância de um requisito de teste e de requisito de teste revelador de erro. As heurísticas utilizadas no estudo empírico são apresentadas na Seção 4.3. A Seção 4.4 descreve as medidas de comparação utilizadas. O estudo de caso é apresentado na Seção 4.5. A discussão dos resultados está na Seção 4.6; e, finalmente, na Seção 4.7, são apresentadas as considerações finais deste capítulo.

4.2 Requisitos de Teste Reveladores de Erro

Nesta seção, os conceitos de *instância* de um requisito de teste e de *requisitos de teste reveladores de erro* são definidos; porém, antes de apresentá-los, alguns conceitos auxiliares são introduzidos.

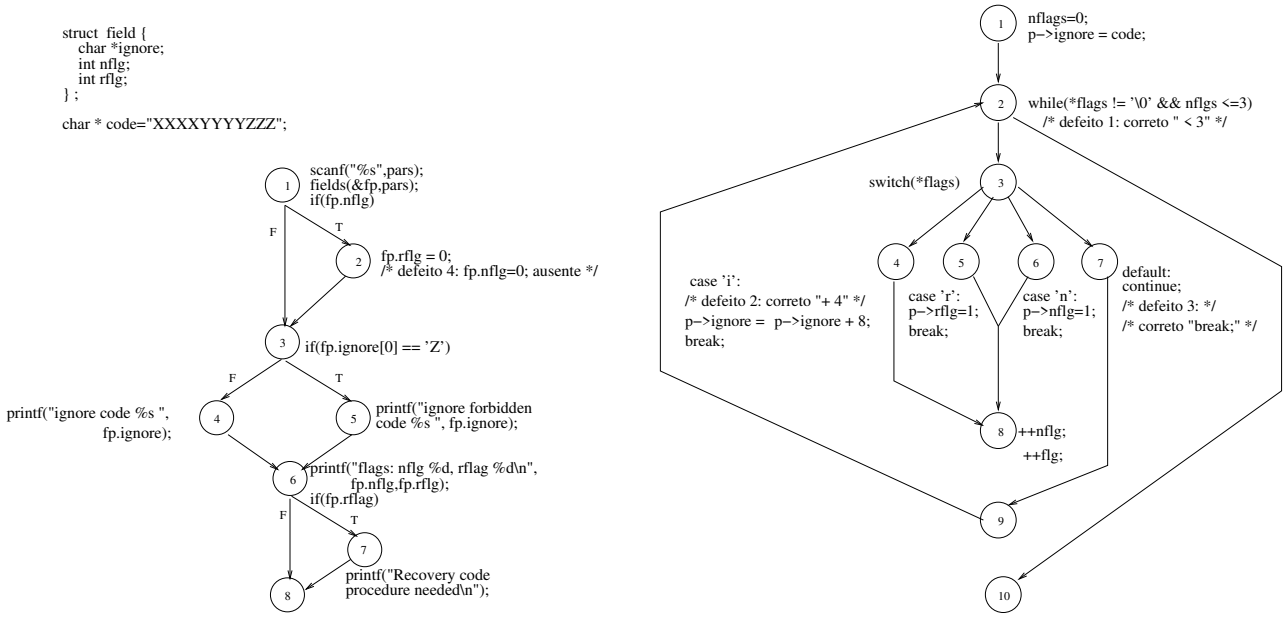
Considere-se um software $\mathcal{S}$ composto de programas P que representam procedimentos ou funções. Seja $P = \{S_1 \ \ldots \ S_i \ \ldots \ S_n\}$, $1 \leq i \leq n$, um programa tal que $G(N, R, e, s)$ é o grafo de fluxo de controle associado a P onde N é o conjunto de blocos de comandos (*nós*) tal que uma vez executado o primeiro comando do bloco todos são executados seqüencialmente, R é o conjunto de ramos, e é o nó de entrada e s é o nó de saída. A *trajetória* de um caso de teste t é dada pelo conjunto $\mathcal{T}$ definido da seguinte maneira: $\mathcal{T} = \{\ i^p \mid i \in N$ ocorre no *caminho completo* CC executado por t e *p* é a posição de *i* em $CC\ \}$. Considerem-se os procedimentos do programa exemplo 3 descritos na Figura 4.1 cujos grafos de fluxo de controle são apresentados na Figura 4.2. Para um caso de teste com entrada igual à cadeia de caracteres "irrn" (referenciado como caso de teste 1), o caminho completo para a função `fields()` é dado por 1, 2, 3, 4, 8, 9, 2, 3, 5, 8, 9, 2, 3, 5, 8, 9, 2, 3, 6, 8, 9, 2, 10. Por sua vez, a *trajetória* $\mathcal{T}$ do caso de teste 1 para a função `fields()` é dada por $1^1, 2^2, 3^3, 4^4, 8^5, 9^6, 2^7, 3^8, 5^9, 8^{10}, 9^{11}, 2^{12}, 3^{13}, 5^{14}, 8^{15}, 9^{16}, 2^{17}, 3^{18}, 6^{19}, 8^{20}, 9^{21}, 2^{22}, 10^{23}$; para a função `main()`, o caminho completo é 1, 2, 3, 4, 6, 8 e a trajetória é $1^1, 2^2, 3^3, 4^4, 6^5, 8^6$.

A partir de agora, i^p é referenciado como uma *instância* do nó i. Note-se que os conceitos de *instância* de um nó e *ponto de execução* (Subseção 2.2) são semelhantes; porém, o primeiro agrega um conjunto de pontos de execução. Sejam X o primeiro comando do nó i e Y o último. A *entrada* da instância i^p é o ponto de execução imediatamente anterior ao ponto X^p; e a *saída* é o ponto de execução imediatamente posterior a Y^p.

A localização do defeito no programa é definida pelo conceito de *sítio do defeito* descrito a seguir. Para as definições que se seguem, considere-se uma linguagem do estilo Algol (Ghezzi e Jazayery, 1987) que possui os seguintes recursos: operadores matemáticos, de atribuição e para manipulação de variáveis

```
#include <stdio.h>

struct  field {
    char *ignore;
    int nflg;
    int rflg;
} ;

char * code="XXXXYYYYZZZ";

fields(p,flgs)
struct field *p;
char * flgs;
{
int nflgs;

nflgs=0;
p->ignore=code;

while (*flgs != '\0' && nflgs <= 3) /* defeito 1: correto "<3" */
   {
   switch(*flgs)
     {
     case 'i': p->ignore= p->ignore + 8;  /* defeito 2: correto "+ 4" */
               break;
     case 'r': p->rflg=1; break;
     case 'n': p->nflg=1; break;
     default: continue;  /* defeito 3: correto "break;" */
     }
   ++nflgs;
   ++flgs;
   }
}

main()
{
char pars[100];
struct field fp;
scanf("%s",pars);

fields(&fp,pars);

if (fp.nflg)
      {
      fp.rflg = 0;
      /* defeito 4: fp.nflg=0; ausente */
     }

if(fp.ignore[0] == 'Z')
  printf("ignore forbidden code %s ",fp.ignore);
else
  printf("ignore code %s ",fp.ignore);

printf("flags: nflg %d, rflg %d\n",fp.nflg, fp.rflg);

if (fp.rflg)
      printf("Recovery code procedure needed\n");
}
```

Figura 4.1: Programa exemplo 3.



(a) `main()` (b) `fields()`

Figura 4.2: Grafos de fluxo de controle das funções `main()` e `fields()`.

escalares, agregadas (e.g., vetores e matrizes) e derreferenciadas (posições de memória endereçadas utilizando ponteiros); comandos de entrada e saída; e comandos de controle de fluxo condicional (e.g., **if**, **case while**, **for**) e incondicional (e.g., **goto**). Nos exemplos apresentados a seguir, é utilizado o jargão da linguagem C.

Definição 1 Seja S_i um comando de um programa P. O comando S_i é o *sítio de um defeito* nas seguintes situações:

1. S_i é definido pela expressão expr(*x,f(y),...*) que contém:
   (a) um operador incorreto; ou
   (b) um operando incorreto (constante ou variável); ou
   (c) uma invocação incorreta de função; ou
   (d) um operando adicional (constante ou variável); ou
   (e) um operando ausente (constante ou variável).
2. S_i é um comando de controle de fluxo incorreto;
3. S_i é um comando não-operacional que representa um comando ausente.

As situações descritas na Definição 1 representam uma classe de defeitos que é uma extensão da classificação de defeitos desenvolvida por Shimomura et al. (1995). Na Figura 4.1, os defeitos 1 e 2 exemplificam a situação 1 e o defeito 3 exemplica a situação 2 da Definição 1. A situação 3 modela o tipo de defeito conhecido como *comando ausente* (DeMillo e Mathur, 1995). O comando fictício não-operacional S_i representa o ponto do programa onde o comando ausente deveria ser inserido para corrigir o programa, como pode ser observado no defeito 4.

Para as próximas definições, são assumidas as seguintes premissas com relação aos nós i, j, m, n, n' pertencentes ao grafo de fluxo $G(N, R, e, s)$ obtido a partir do programa P: o sítio do defeito está localizado em um comando do programa P que foi mapeado para o nó i; o nó m possui dois sucessores, n e n', sendo que o nó n *domina*[1] o nó j, mas n' não; e o último comando do nó m é um comando de controle de fluxo condicional cujo predicado é expr(x,*f(y)*,...). Considere-se também que $\mathcal{T} = 1^1, \ldots, i^p, \ldots, m^q, n^{q+1}, j^r, k^{r+1}, \ldots$ é a trajetória de um caso de teste revelador de defeito t, onde $1 \leq p \leq q \leq r$.

Definição 2 O par (m^q,n^{q+1}) é definido como uma *instância* do *ramo* (m, n). Define-se, por sua vez, como *instância* de um requisito de teste de fluxo de dados (*adu* ou *adpu*) (i, j, D) ou $(i, (j, k), D)$[2], a quíntupla $(i^p, j^r, D, \mathcal{V}, \mathcal{V}')$ ou $(i^p, (j^r, k^{r+1}), D, \mathcal{V}, \mathcal{V}')$ tal que:

1. a trajetória $i^p, \ldots, j^r$ (para (i, j, D)) ou $i^p, \ldots, j^r$, k^{r+1} (para $(i, (j, k), D)$) é livre de definição c.r.a. variáveis $v \in D$;
2. $\mathcal{V}$ é o conjunto de valores de $v \in D$ na saída de i^p;
3. $\mathcal{V}'$ é o conjunto de valores de $v \in D$ na entrada de j^r (para (i, j, D)) ou k^{r+1} (para $(i, (j, k), D)$).

Os valores contidos em $\mathcal{V}$ e $\mathcal{V}'$ podem ser diferentes devido a efeitos colaterais ou ao modelo de dados utilizado. Os valores nos nós de *definição* e de *uso* de uma variável `v` podem ser diferentes porque o valor de `v` é alterado erroneamente na trajetória da instância ou porque as variáveis derreferenciadas e as variáveis agregadas (e.g., vetores e matrizes) são tratadas como um objeto único de memória de acordo com o modelo de dados (Horgan e London, 1991; Vilela et al., 1997). Nesta segunda situação, suponha-se que o elemento `v[0]` no nó de definição receba o valor 2 e que o elemento referenciado no nó de uso seja `v[10]` cujo valor é 12. Note-se que os valores são diferentes devido a forma de tratamento de vetores no modelo de dados utilizado.

A Tabela 4.1 contém os requisitos de teste estabelecidos pelos critérios todos usos para o programa exemplo 3 da Figura 4.1. Inspecionando a trajetória do caso de teste 1 na função `fields()`, pode-se

[1] Um nó n *domina* (ou é *dominador* de) um outro nó j se e somente se todo caminho do nó de entrada até j inclui n (Hecht, 1977).

[2] D é o conjunto de variáveis definidas no nó i (Subseção 2.3.1), sendo que, para os critérios todos p-usos e todos usos, $|D| = 1$ e, para os critérios Potenciais Usos, $|D| \geq 1$.

Tabela 4.1: Requisitos de teste dos critérios todos usos para os procedimentos `main()` e `fields()`.

`main()`		`fields()`	
1	(1,(3,4), fp.ignore)	1	(1,4, p)
2	(1,(3,5), fp.ignore)	2	(1,5, p)
3	(1,4, fp.ignore)	3	(1,6, p)
4	(1,5, fp.ignore)	4	(1,4, p->ignore)
5	(1,(3,4), *(fp.ignore))	5	(1,(2,3), flgs)
6	(1,(3,5), *(fp.ignore))	6	(1,(2,10), flgs)
7	(1,4, *(fp.ignore))	7	(1,(3,4), flgs)
8	(1,5, *(fp.ignore))	8	(1,(3,5), flgs)
9	(1,(1,2), fp.nflg)	9	(1,(3,6), flgs)
10	(1,(1,3), fp.nflg)	10	(1,(3,7), flgs)
11	(1,6, fp.nflg)	11	(1,8, flgs)
12	(1,6, fp.rflg)	12	(1,(2,3), nflgs)
13	(1,(6,7), fp.rflg)	13	(1,(2,10), nflgs)
14	(1,(6,8), fp.rflg)	14	(1,8, nflgs)
15	(2,6, fp.rflg)	15	(4,4, p->ignore)
16	(2,(6,7), fp.rflg)	16	(8,(2,3), flgs)
17	(1,(6,8), fp.rflg)	17	(8,(2,10), flgs)
		18	(8,(3,4), flgs)
		19	(8,(3,5), flgs)
		20	(8,(3,6), flgs)
		21	(8,(3,7), flgs)
		22	(8,8, flgs)
		23	(8,(2,3), nflgs)
		24	(8,(2,8), nflgs)
		25	(8,8, nflgs)

observar que o ramo `(2,3)` e a *adu* `(7, (2,3), nflags)` possuem quatro e três instâncias, respectivamente: $(2^2,3^3)$, $(2^7,3^8)$, $(2^{12},3^{13})$, e $(2^{17},3^{18})$; e $(8^5,(2^7,3^8)$,nflgs $, 1, 1)$, $(8^{10},(2^{12},3^{13})$,nflgs $, 2, 2)$, e $(8^{15},(2^{17},3^{18})$,nflgs, $3, 3)$.

Definição 3 Um instância j^r de um nó j é *incorretamente alcançada* se:

1. existe uma instância (m^q,n^{q+1}) do ramo (m, n) que é executada devido a um resultado incorreto de expr(x,*f(y)*,...) em m^q causado pelo prévio alcance do sítio do defeito em i^p, $p \leq q$; ou

2. o sítio do defeito é um comando incondicional que foi alcançado em i^p, $p < r$, e que provocou a ocorrência da instância j^r.

As situações acima descrevem as possíveis causas imediatas do alcance de um ponto do programa que não deveria ser atingido durante a execução. Considerando novamente a função `fields()` e o caso de teste 1, a instância 6^{19} é *incorretamente alcançada* porque $(2^{17}, 3^{18})$ ocorre devido ao resultado incorreto do predicado `(*flgs != '\0' && nflgs <= 3)` em 2^{17} que, por sua vez, é conseqüência do

sítio do defeito ter sido alcançado no nó 2. Note-se ainda que, como o nó 3 domina o nó 6, a instância $(2^{17}, 3^{18})$ *provoca* a ocorrência da instância 6^{19}.

Nas definições seguintes, as mesmas premissas estabelecidas antes da Definição 2 continuam válidas com exceção daquela que considera o sítio do defeito localizado no nó i.

Definição 4 Diz-se que uma instância (m^q,n^{q+1}) do ramo (m, n) é executada devido a um *efeito do defeito* quando uma das seguintes situações ocorrem:

1. expr(x,*f(y)*,...) em m^q produz um resultado incorreto que causa a execução de n^{q+1}, sendo que o resultado incorreto ocorre porque:
   (a) expr(x,*f(y)*,...) é o sítio do defeito ou
   (b) a variável x possui um valor incorreto em m^q devido ao prévio alcance do sítio do defeito ou
   (c) o valor da variável y está correto; porém, f(y) retorna um valor incorreto devido ao prévio alcance do sítio do defeito;
2. a instância m^q é alcançada incorretamente.

Na função `fields()`, a instância (2^{17},3^{18}) ocorre devido a um efeito do defeito porque seu sítio está localizado no nó 2 (situação 1a da Definição 4). Já (6^{19},8^{20}) ocorre porque a instância 6^{19} foi incorretamente alcançada (situação 2 da Definição 4). A instância (3^3,4^4) na funcão `main()`, por sua vez, é executada devido a um efeito do defeito descrito na situação 1b da Definição 4, isto é, o resultado incorreto do predicado `fp.ignore[0] == 'Z'` é causado pelo valor incorreto da variável `fp.ignore` devido, por sua vez, ao alcance prévio do sítio do defeito.

Para a definição a seguir, seja $S : v =$ expr'(x,f(y),...) o último comando do nó i que atribui um valor à variável v.

Definição 5 O valor de v na saída de i^p é definido como incorreto devido a um *efeito do defeito* se uma das seguintes situações ocorrem:

1. a expressão $v =$ expr'(x,f(y),...) é o sítio do defeito; ou
2. a variável x possui um valor incorreto em i^p devido ao alcance prévio do sítio do defeito; ou
3. o valor da variável y está correto; porém, f(y) retorna um valor incorreto devido ao alcance prévio do sítio do defeito; ou
4. a instância i^p é alcançada incorretamente.

Para a próxima definição, considere-se S : expr(v,f(w),...) o primeiro comando do nó j onde há uma referência a v.

Definição 6 O valor de v na entrada de j^r é definido como incorreto devido a um *efeito do defeito* se uma das seguintes situações ocorrer:

1. o sítio do defeito foi previamente alcançado e o valor de v em j^r é ν' quando deveria ser ν;
2. a instância j^r foi incorretamente alcançada.

Os exemplos utilizados para ilustrar a Definição 4 podem também ser usados para exemplificar a maioria das situações relativas às Definições 5 e 6. A exceção é a situação 1 da Definição 6. Considere-se a referência à variável `fp.nflg` no nó 6 da função `main()`, o valor de `fp.nflg` em 6^5 é 1, mas deveria ser 0. O valor de `fp.nflg` não mudou, quando deveria, porque o sítio do defeito foi alcançado em 2^2.

A partir do conceito de *efeito do defeito*, são definidos os *requisitos de teste reveladores de erros* (*rt-re*).

Definição 7 Um ramo (m, n) é *revelador de erro* se:

1. existe uma instância (m^q, n^{q+1}) exercitada em um caso de teste $t \in T_R$ devido a um efeito do defeito; ou
2. a instância (m^q, n^{q+1}) alcança o sítio do defeito.

Definição 8 Uma *adu* (i, j, v) é *reveladora de erro* se existe uma instância (i^p, j^r, v, ν, ν') exercitada por um caso de teste $t \in T_R$ onde:

1. o valor de v na saída de i^p é incorreto devido a um efeito do defeito; ou
2. o valor de v na entrada de j^r é incorreto devido a um efeito do defeito; ou
3. a execução de j^r faz com que o sítio do defeito seja alcançado.

Definição 9 Uma *adu* ou *adpu* $(i, (j, k), D)$ é *reveladora de erro* se existe uma instância $(i^p, (j^r, k^{r+1}), D, \mathcal{V}, \mathcal{V}')$ exercitada por um caso de teste $t \in T_R$ onde:

1. existe $v \in D$ tal que o valor de v na saída de i^p é incorreto devido a um efeito do defeito; ou
2. existe $v \in D$ tal que o valor de v na entrada de k^{r+1} é incorreto devido a um efeito do defeito; ou
3. a instância (j^r,k^{r+1}) do ramo (j,k) é executada devido a um efeito do defeito; ou

4. a execução de k^{r+1} faz com que o sítio do defeito seja alcançado.

Considerem-se os seguintes exemplos de *requisitos de teste reveladores de erro* obtidos do exemplo da Figura 4.1. O ramo (2,3) em `fields()` é revelador de erro porque a instância (2^{17},3^{18}) é executada devido a um efeito do defeito 1 (situação 1 da Definição 7); e o ramo (1,2) em `main()` é revelador de erro porque a instância (1^1,2^2) *alcança* o sítio do defeito 4, embora a sua execução não seja causada pelo alcance prévio de nenhum outro defeito (situação 2 da Definição 7).

A *adu* `(4,4,p->ignore)` em `fields()` é reveladora de erro porque em qualquer uma de suas instâncias a situação 1 da Definição 8 ocorre. Na verdade, a situação 1 ocorre porque o sítio do defeito está localizado no nó 4, o que faz o valor de `p->ignore` incorreto devido a um efeito do defeito 2. A *adu* `(1,6,fp.nflg)` em `main()` é reveladora de erro porque na instância (1^1,6^5,fp.nflg,1,1) o valor da variável `fp.nflg` na entrada de 6^5 é incorreto visto que há um definição ausente no caminho 1^1, ...,6^5, o que implica a situação 2 da Definição 8.

A *adu* `(1,(3,4),fp.ignore)`[3] em `main()` é também reveladora de erro por duas razões presentes na sua instância (1^1,(3^3,4^4),fp.ignore, 0x80485a8, 0x80485a8): o valor de `fp.ignore` é incorreto em 1^1 (situação 1 da Definição 9); e a instância (3^3,4^4) não deveria ter sido executada (situação 3 da Definição 9). Ambas as situações são conseqüência do efeito do defeito 2 (alcançado em `fields()`) que se propagou para `fp.ignore` através da passagem de parâmetros por referência da variável `fp` na invocação da função `fields()` em `main()`. Já a *adu* `(1,(1,2),fp.nflg)` em `main()` possui uma única instância — (1^1,(1^1,2^2),fp.nflg,1,1) — cujos valores de `fp.nflg` em 1^1 e 2^2 estão corretos e a instância (1^1,2^2) é corretamente executada. Mesmo assim essa *adu* é *reveladora de erro* porque quando ela é exercitada o sítio do defeito 4 é alcançado em 2^2 (situação 4 da Definição 9).

4.3 Heurísticas para Selecionar Requisitos de Teste

Em um cenário ideal, o mantenedor deveria ser capaz de investigar todos os requisitos exercitados por um caso de teste revelador de defeito para determinar aqueles que fornecem informação útil para a depuração. Entretanto, o número de requisitos exercitados por um único caso de teste pode ser muito grande dependendo do critério utilizado. Por exemplo, um único caso de teste submetido ao programa `Sort` exercita 1207 *adu*s de um total de 2398 requeridas pelo critérios todos usos (utilizando o modelo de fluxo de dados **nível 1**).

Para reduzir o número de requisitos a serem investigados a um número tratável, heurísticas são utilizadas. As heurísticas foram propostas para identificar um pequeno número de linhas de código com grande probabilidade de conter o defeito. No estudo de caso realizado, o objetivo das heurísticas foi identificar não somente linhas de código, mas um subconjunto pequeno de requisitos de teste com grande

[3]Como os requisitos de teste do programa da Figura 4.1 foram determinados utilizando modelo de fluxo de dados **nível 1** (Subseção 3.2.2), os campos de `fp` são possivelmente definidos na chamada da função `fields()` no nó 1 de `main()`.

chance de revelar erros. Este subconjunto identificado utilizando heurísticas é chamado de *requisitos de teste candidatos a revelar erros* (*rt-c-re*).

A seguir, são descritas as heurísticas utilizadas no estudo de caso para identificar *rt-c-re*s:

H1 — selecione os requisitos de teste r exercitados por todo caso de teste $t \in T_R$, isto é, os requisitos r presentes no conjunto $\bigcap RT_C(t)$.

Esta heurística é classificada como do tipo *fatiamento*, pois inspeciona requisitos exclusivamente exercitados por todos casos de teste de T_R. Sua motivação é que o subconjunto de requisitos identificado possui mais chances de incluir o defeito ou ser revelador de erro por ser comum a todos os casos de teste reveladores de defeito. Pan e Spafford (1992) foram quem primeiramente definiram esta heurística utilizando *fatias* de programas.

H2 — escolha um caso de teste $t \in T_R$ revelador de defeito e outro $t' \in T_{NR}$ não revelador de defeito e selecione os requisitos de teste r que pertencem a $RT_C(t)$ mas não a $RT_C(t')$, isto é, os requisitos r presentes no conjunto $RT_C(t) - RT_C(t')$.

Esta heurística é classificada como do tipo *recorte* e inspeciona os requisitos que pertencem ao resultado da subtração dos requisitos exercitados por um caso de teste não-revelador de defeito do conjunto de requisitos exercitados por um caso de teste revelador de defeito. O mantenedor pode selecionar os casos de teste t e t' a partir do seu conhecimento do programa; porém, no estudo de caso, t e t' foram selecionados aleatoriamente supondo que o mantenedor não tivesse nenhum conhecimento do programa. A idéia de subtração de artefatos para localização de defeitos já foi explorada em vários trabalhos anteriores (Agrawal, 1991; Chen e Cheung, 1997; Collofello e Cousins, 1987; Lyle e Weiser, 1987; Pan e Spafford, 1992).

Para a definição das heurísticas do tipo *ranking* descritas a seguir, considere-se o conjunto de casos de teste reveladores de defeito $T_{R_r} = \{t \in T_R | r \in RT_C(t)\}$ e o conjunto de casos de teste não-reveladores de defeito $T_{NR_r} = \{t \in T_{NR} | r \in RT_C(t)\}$ que exercitam um requisito r em particular. As heurísticas abaixo atribuem a cada requisito de teste um *peso* (p_r) definido pela seguinte fórmula: $p_r = |T_{R_r}|/(1 + |T_{NR_r}|/|T|)$.

De acordo com esta fórmula, os requisitos de teste mais bem classificados são aqueles freqüentemente exercitados por casos de teste reveladores de defeito e não tão freqüentemente por casos de teste não-reveladores de defeito. O objetivo da classificação é filtrar os requisitos de testes *pouco* relacionados com o comportamento errôneo do programa, realçando aqueles *bastante* relacionados.

H3 — ordene decrescentemente, de acordo com o peso p_r, os requisitos r e selecione aqueles classificados em primeiro lugar.

H4 — ordene decrescentemente, de acordo com o peso p_r, os requisitos r e selecione aqueles classificados em primeiro e segundo lugares.

Na fórmula para cálculo do peso p_r do requisito r, duas situações extremas podem ocorrer — $T_{R_r} = \phi$ e $T_{NR_r} \neq \phi$; e $T_{R_r} \neq \phi$ e $T_{NR_r} = \phi$. Na primeira situação, $p_r = 0$, pois o requisito r não foi executado por casos de teste reveladores de defeito; logo, ele não pode ser revelador de erro e possui baixa probabilidade de incluir o defeito no seu trecho de código associado. Na segunda situação, $p_r = |T_{R_r}|$ visto que $|T_{NR_r}| = 0$. Note-se, todavia, que esta situação não garante que r será escolhido, pois outros requisitos r' podem possuir $p_{r'} > p_r$ mesmo que tenham sido executados por casos de teste não-reveladores de defeito (i.e., $T_{NR_r} \neq \phi$).

A idéia de classificar os requisitos de teste foi desenvolvida por Collofello e Cousins (1987) e também utilizada por Pan e Spafford (1992). No entanto, a fórmula acima é diferente da utilizada nos trabalhos anteriores porque permite um ajuste mais fino dos pesos atribuídos aos requisitos de teste. As duas heurísticas do tipo *ranking* (H3 e H4) foram selecionadas para analisar o efeito da expansão da classificação na localização dos defeitos.

4.4 Medidas de Comparação

Medidas de *inclusão*, *eficiência* e *custo* foram utilizadas para avaliar o uso de informação de teste na depuração por meio de heurísticas. Esses três aspectos foram avaliados com respeito à aplicação de uma heurística H usando os requisitos de teste estabelecidos por um critério C sobre conjuntos de casos de teste T_i, $1 < i \leq n$. As medidas foram definidas considerando que o sítio do defeito está localizado em um comando S e utilizando os seguintes conjuntos:

- $E_{(C,T_i)}$ – conjunto de requisitos de teste reveladores de erros exercitados por casos de teste reveladores de defeito t pertencentes a T_i;
- $CE_{(H,C,T_i)}$ – conjunto de requisitos candidatos a reveladores de erros selecionados pela aplicação de (H, C) no conjunto T_i; e
- $Cmd_{(H,C,T_i)}$ – conjunto de comandos associados aos requisitos r pertencentes a $CE_{(H,C,T_i)}$.

4.4.1 Inclusão

A medida de inclusão avalia a habilidade do par (H, C) em fornecer informação útil para a localização do defeito. Dois tipos de informação são fornecidas: o conjunto de requisitos candidatos a revelar erro ($CE_{(H,C,T_i)}$) e o trecho de código associado a esses requisitos ($Cmd_{(H,C,T_i)}$).

A utilidade da informação fornecida pode ser avaliada verificando se os requisitos de teste selecionados incluem pelos menos um requisito que seja revelador de erro (*rt-re*) ou se o trecho de código identificado inclui o sítio do defeito. A segunda forma de avaliação da inclusão foi utilizada em trabalhos anteriores (Agrawal et al., 1995; Collofello e Cousins, 1987; Pan e Spafford, 1992); porém, ela é muito

limitada, pois não leva em consideração outras indicações úteis para a localização do defeito fornecidas pelos requisitos de teste. Para efeitos de comparação, foram definidas duas medidas, uma considerando a inclusão de *rt-re*s e a outra a inclusão do sítio do defeito.

Definição 10 A *Taxa de Inclusão do Sítio do Defeito* ($Inc_{\text{sítio}}$) é a razão entre o número de conjuntos $Cmd_{(H,C,T_i)}$ que acertam o sítio do defeito e o número total de conjuntos $Cmd_{(H,C,T_i)}$, definida como se segue:

$$Inc_{\text{sítio}} = \frac{|\{Cmd_{(H,C,T_i)} | S \in Cmd_{(H,C,T_i)}\}|}{n}, \text{onde } 0 < i \leq n.$$

Definição 11 A *Taxa de Inclusão de Requisitos Reveladores de Erro* (Inc_{erro}) é a razão entre o número de conjuntos $CE_{(H,C,T_i)}$ que incluem pelo menos um *rt-re* e o o número total de conjuntos $CE_{(H,C,T_i)}$, definida como se segue:

$$Inc_{\text{erro}} = \frac{|\{CE_{(H,C,T_i)} | \exists r \in CE_{(H,C,T_i)} \wedge r \in E_{(C,T_i)}\}|}{n}, \text{onde } 0 < i \leq n.$$

Vários fatores podem influenciar os resultados das medidas de inclusão, a saber, a posição do defeito no código fonte, as variáveis que ele afeta etc. A análise dos resultados do estudo de caso indica que esses fatores em geral afetam o resultado do par (H, C) em conjunto. Por esta razão, a combinação dos fatores relacionados com o defeito em si é chamada de *idiossincrasia do defeito*.

4.4.2 Eficiência

A medida de *eficiência* visa verificar a habilidade do par (H, C) em reduzir o espaço de busca para a localização do defeito. Esta medida também pode ser avaliada de duas maneiras: comparando o tamanho do conjunto $Cmd_{(H,C,T_i)}$ com o tamanho de um conjunto de referência (e.g., todos comandos do programa) para determinar se ocorreu uma redução significativa do espaço de busca do defeito; ou comparando o número de *rt-re*s presentes no conjunto $CE_{(H,C,T_i)}$ com o número total de elementos de $CE_{(H,C,T_i)}$.

Definição 12 A *Eficiência em termos de Comandos* (Ef_{cmd}) é definida como a média de redução, em termos de comandos a serem investigados, proporcionada pelos conjuntos $Cmd_{(H,C,T_i)}$ em comparação com o conjunto de referência escolhido quando o sítio do defeito está contido em $Cmd_{(H,C,T_i)}$. Ef_{cmd} é descrita pela fórmula a seguir:

$$Ef_{\text{cmd}} = \frac{\sum_1^n Ef_{\text{cmd}_{T_i}}}{n \times Inc_{\text{sítio}}}, \text{onde } Ef_{\text{cmd}_{T_i}} = \begin{cases} \frac{|Cmd_{(H,C,T_i)}|}{\text{tamanho do conjunto de referência}} & \text{se } S \in Cmd_{(H,C,T_i)}, \\ 0 & \text{caso contrário.} \end{cases}$$

Definição 13 A *Eficiência em termos de Requisitos Reveladores de Erros* (Ef_{erro}) é definida como a porcentagem média de elementos dos conjuntos $CE_{(H,C,T_i)}$ que são reveladores de erro quando pelo menos um *rt-re* está contido em $CE_{(H,C,T_i)}$. Ef_{erro} é descrita pela fórmula a seguir:

$$Ef_{\text{erro}} = \frac{\sum_1^n Ef_{\text{erro}_{T_i}}}{n \times Inc_{\text{erro}}}, \text{onde } Ef_{\text{erro}_{T_i}} = \frac{|\{r | r \in CE_{(H,C,T_i)} \wedge r \in E_{(C,T_i)}\}|}{|CE_{(H,C,T_i)}|}$$

Note-se que as medidas de eficiência levam em consideração somente as aplicações bem sucedidas do par (H, C). Os denominadores de Ef_{cmd} e Ef_{erro} são, respectivamente, $n \times Inc_{\text{sítio}}$ e $n \times Inc_{\text{erro}}$ para que a média seja calculada em função do número de vezes em que o par (H,C) aplicado a um conjunto T_i identifica o sítio do defeito ou seleciona pelo menos um *rt-re*. Isto ocorre porque a redução no espaço de busca somente tem sentido se o sítio do defeito ou pelo menos um *rt-re* estiver presente pois, caso contrário, o mantenedor irá procurar o defeito no *lugar errado* e nunca o localizará. Neste sentido, $Inc_{\text{sítio}}$ e Inc_{erro} indicam a probabilidade do alvo (sítio do defeito ou *rt-re*) ser atingido enquanto Ef_{cmd} e Ef_{erro} dão uma estimativa do esforço remanescente para localização do defeito quando o alvo é atingido.

Em termos de análise do espaço de busca, quanto menor for o valor de Ef_{cmd}, melhor é o desempenho do par (H,C); já Ef_{erro} é melhor quanto mais próximo de 100% estiver, pois isto significa que qualquer requisito selecionado é revelador de erro. Com relação a Ef_{cmd}, um aspecto importante é o *conjunto de referência*. A escolha óbvia seria o programa todo; entretanto, qualquer par (H,C) gera um conjunto $Cmd_{(H,C,T_i)}$ que é menor ou igual ao programa todo. Dessa forma, é desejável que o conjunto de referência de comparação seja menor que o programa todo e possua uma grande probabilidade de incluir o defeito.

4.4.3 Custo

Dado um conjunto de casos de teste T_i, o custo de obtenção do conjunto $CE_{(H,C,T_i)}$ é determinado por dois componentes principais: *análise de adequação* dos casos de teste de T_i com respeito ao critério de teste C e *processamento das heurísticas*.

O custo da análise de adequação (C_4) para *um* caso de teste foi discutido na Subseção 3.2.4 e é definido por $O(|RT_C|)$. Logo, o custo de fazer a análise de adequação de todos casos de teste de T_i é $O(|T_i| \times |RT_C|)$.

Com relação ao processamento das heurísticas, o custo principal ocorre quando são lidas as informações estáticas e dinâmicas relativas aos requisitos de teste. Isto porque as operações implementadas pelas heurísticas possuem um custo muito pequeno em relação ao de leitura, pois foram implementadas de forma eficiente. As operações com conjuntos (subtração e interseção) das heurísticas H1 e H2 são realizadas utilizando vetores de bits e a operação de classificação das heurísticas H3 e H4 é implemen-

tada utilizando o algoritmo *quick sort*. Portanto, o custo de processamento das heurísticas é também $O(|T_i| \times |RT_C|)$ visto que é necessário ler as informações relativas a cada requisito de teste para todos os casos de teste de T_i.

Assim, o custo de aplicação de uma heurística H utilizando os requisitos de teste de um critério C é $O(|T_i| \times |RT_C|)$. Dessa forma, o custo empírico de aplicação de (H, C) pode ser definido como se segue:

Definição 14 O *custo empírico* de aplicação de (H, C) em conjuntos de casos de teste T_i, $1 < i \leq n$, é dado por:

$$\text{Custo}_{(H,C)} = \frac{\sum_1^n |T_i| \times |RT_C|}{n} = \text{média}(|T_i|) \times |RT_C|.$$

4.5 Estudo de Caso

4.5.1 Descrição do Experimento

Programas Utilizados

Neste estudo de caso, as versões *defeituosas* do programa `Sort` desenvolvidas por Wong (1993) foram novamente utilizadas (Subseção 3.3.1). Os programas objeto do estudo de caso foram as onze versões cujos defeitos eram mais difíceis de detectar determinadas por Delamaro et al. (2001a). A Tabela 4.2 descreve os defeitos contidos nas versões utilizadas e o índice de dificuldade de cada uma delas.

Cenários de Teste-Depuração

O experimento apresentado a seguir simula três cenários de teste-depuração em que os requisitos de teste de um critério estrutural são utilizados para auxiliar a localização de defeitos por meio de heurísticas. Dois cenários que ocorrem tipicamente durante o desenvolvimento e um que ocorre durante a manutenção são simulados.

Os cenários visam caracterizar como a atividade de teste é conduzida e quais as informações disponíveis ao mantenedor durante a depuração. Em quaisquer dos cenários, os resultados da análise de adequação estão disponíveis; porém, a maneira como é conduzida a atividade de teste determina as características dos conjuntos de casos de teste a serem utilizados na depuração. Algumas das possíveis características desses conjuntos são: ser adequado ou não ao critério de teste; se não-adequado, possuir alta ou baixa cobertura em relação ao critério; possuir um único ou vários casos de teste reveladores de defeito; etc.

Tabela 4.2: Defeitos contidos nas versões do programa `Sort`

Defeito	Código Original	Código Incorreto	ID
1	`while(...ip[-1]->l<0)`	`while(...ip[-1]->l<=0)`	3,99
2	`if (pb < lb && *pb==tabchar)`	`if (pb < lb || *pb==tabchar)`	4,63
3	`c == ' ' || c == '\t'`	`c == ' ' && c == '\t'`	25,20
4	`p->ignore = dict+128;`	`p->ignore = dict+290;`	7,17
5	`break;`	`continue;`	5,57
6	`for(k=lp+1; k<=hp;)`	`for(k=lp+1; k<hp;)`	13,20
7	`if(b= *--ipb - *--ipa)`	`if(b= *--ipb - --*ipa)`	10,60
8	`while((*dp++ = *cp++) != '\n');`	`while((*++dp = *cp++) != '\n');`	13,77
9	`if(*--ipa != '0')`	`if(--*ipa != '0')`	11,53
10	`*pb == '\n' ? -fields[0].rflg`	`*pb == '\n' ? -fields[0].nflg`	23,23
11	`while (++k < j)`	`while (++j < k)`	2,87

O primeiro cenário — referenciado como *cenário testador-programador* (TP-C) — ocorre naquelas organizações em que o programador é responsável pelo teste e pela depuração. Neste cenário, o programador cria os casos de teste um por vez. Cada caso de teste é então executado, validado em relação à especificação do programa e analisado quanto a sua adequação ao critério escolhido. Se um caso de teste revelador de defeito é encontrado, então o processo de teste pára e a atividade de depuração começa tendo como subsídio a informação coletada até então.

O segundo cenário — chamado de *cenário grupo de teste* (GT-C) — pressupõe organizações em que existe um grupo independente responsável pela atividade de teste, embora o programador ainda seja responsável pela depuração do código fonte. Os casos de teste são gerados, submetidos, validados e analisados pelo grupo de teste independente até que um conjunto adequado é obtido. O programa é retornado ao programador para depuração, juntamente com as informações coletadas, se um ou mais casos de teste são reveladores de defeito.

Finalmente, o *cenário de manutenção* (MT-C) trata as situações em que o programador deve depurar o programa depois de receber um caso de teste revelador de defeito obtido depois da liberação do software. Ele pressupõe que foi desenvolvido um conjunto de casos de teste adequado a um critério estrutural durante a atividade de teste. Os resultados coletados durante o teste mais as informações obtidas da análise de adequação do novo caso de teste são fornecidas ao mantenedor para depuração.

Simulando os Cenários

A massa de casos de teste desenvolvida por Wong (1993) foi utilizada para simular a geração de conjuntos de casos de teste T_i de acordo com os cenários descritos acima. O cenário testador-programador (TP-C) é simulado utilizando um critério estrutural C como guia para a seleção de casos

de teste. Os conjuntos T_i são criados pela seleção aleatória de casos de teste da massa de forma que pelo menos um novo requisito é exercitado pelo caso de teste selecionado. A seleção pára quando o primeiro caso de teste revelador de defeito é descoberto. Neste cenário, todos os conjuntos obtidos são *não-adequados* ao critério C e possuem um único caso de teste revelador de defeito.

A simulação do cenário grupo de teste (GT-C) é realizada selecionado conjuntos T_i *adequados* ao critério C a partir da massa de teste. Os casos de teste são selecionados aleatoriamente e adicionados ao conjunto T_i desde que aumentem a cobertura de T_i em relação ao critério C. Este processo termina quando a máxima cobertura possível é obtida. Neste estudo de caso, os conjuntos de casos de teste que atingem a *máxima cobertura* (obtida utilizando todos os 997 casos de teste da massa de teste) são assumidos como *adequados*. Assim, os conjuntos obtidos são adequados e possuem um ou mais casos de teste reveladores de defeitos.

O cenário de manutenção baseado no critério de teste C (MT-C) utiliza conjuntos T_i derivados da seguinte maneira: casos de teste não-reveladores de defeito são selecionados aleatoriamente e incluídos em T_i desde que aumentem a sua cobertura de acordo com C; a seleção de casos de teste não-reveladores de defeito termina quando a máxima cobertura é obtida ou depois de 1000 tentativas. O limite no número de tentativas é necessário porque para algumas versões do `Sort` todo conjunto adequado inclui pelo menos um caso de teste revelador de defeito e, por isso, apenas conjuntos *quase-adequados* podem ser obtidos. Para completar o cenário, um caso de teste revelador de defeito é selecionado (aleatoriamente) a partir da massa de teste para simular a ocorrência de uma falha depois da liberação do software. Portanto, esses conjuntos possuem um valor de cobertura alto (pois são adequados ou quase-adequados) e contêm um único caso de teste revelador de defeito.

Coletando os Resultados das Heurísticas

Os dados experimentais foram obtidos utilizando o módulo `pokedebugheur` da ferramenta POKE-TOOL que implementa as heurísticas descritas anteriormente. Os pares (H, C), onde $H \in \{$H1, H2, H3, H4$\}$ e $C \in \{$todos ramos, todos usos, todos potenciais usos$\}$, foram aplicados em conjuntos T_i, $1 < i \leq 30$, gerados para cada um dos cenários simulados (TP-C, GT-C e MT-C). Os conjuntos $CE_{(H,C,T_i)}$ foram os resultados coletados, enquanto os conjuntos $Cmd_{(H,C,T_i)}$ foram determinados pela identificação dos comandos associados aos requisitos de teste contidos nos conjuntos $CE_{(H,C,T_i)}$.

A medida $Inc_{\text{sítio}}$ foi calculada verificando se o sítio do defeito estava presente nos conjuntos $Cmd_{(H,C,T_i)}$. A medida Ef_{cmd} foi obtida comparando o tamanho dos conjuntos $Cmd_{(H,C,T_i)}$ com o tamanho dos conjuntos $Cmd_{(H1,C,T_i)}$ (i.e., selecionados utilizando H1). Os conjuntos de H1 foram escolhidos como conjuntos de referência porque sua probabilidade de conter o sítio do defeito é grande e o número de comandos selecionados é menor que o do programa todo. O tamanho médio dos conjuntos $Cmd_{(H,C,T_i)}$ que incluem o sítio do defeito é indicado por $Cmd_{\text{médio}}$.

As medidas Inc_{erro} e Ef_{erro}, por sua vez, foram calculadas de maneira aproximada devido à grande

quantidade de requisitos selecionados por algumas heurísticas. Este fato torna muito difícil a determinação de *todos* os requisitos reveladores de erros (*rt-re*s) selecionados pelas heurísticas. Este processo foi automatizado parcialmente de duas maneiras.

A primeira foi pela identificação (via inspeção do código) daqueles requisitos que são certamente reveladores de erro sempre que são exercitados por um caso de teste que provoca a ocorrência de uma falha. A outra maneira foi pela comparação dos resultados da análise de adequação das versões incorretas do `Sort` com os resultados da versão correta. Isto foi feito da seguinte forma. Para cada versão defeituosa do `Sort`, foi selecionado um caso de teste revelador de defeito para o qual os resultados da análise de adequação das versões defeituosa e correta do `Sort` foram comparados. A comparação restringiu-se àquelas funções cujos requisitos de teste em ambas as versões eram idênticos, isto é, o defeito não alterava a definição dos requisitos de teste. Um requisito de teste era considerado revelador de erro se o resultado da análise de adequação do requisito nas versões correta e defeituosa do `Sort` era diferente com relação ao *número de instâncias* ou à posição na trajetória da *primeira* e *última* instâncias. Note-se que esta análise captura aquelas situações em que alguma instância do requisito ou não deveria ter ocorrido ou foi incorretamente alcançada. Assim, devido à maneira aproximada como foram determinados os *rt-re*s, as medidas Inc_{erro} e Ef_{erro} apresentadas são subestimadas. O tamanho médio dos conjuntos $CE_{(H,C,T_i)}$ que incluem pelo menos um *rt-re* é dado por $CE_{\text{médio}}$.

Os dados relativos às medidas $Inc_{\text{sítio}}$, Ef_{cmd}, Inc_{erro} e Ef_{erro} são apresentados utilizando gráficos. Os valores dessas medidas são apresentados em função das versões defeituosas do programa `Sort`. As Figuras 4.3 a 4.8 contêm os dados relativos ao cenário TP-C; as Figuras 4.9 a 4.14, os dados do cenário GT-C; e as Figuras 4.15 a 4.20, os dados do cenário MT-C.

Os gráficos relativos a $Inc_{\text{sítio}}$ e Inc_{erro} contêm uma *régua* na altura do valor 75% para facilitar a análise dos resultados. Com o mesmo objetivo, Ef_{cmd} e Ef_{erro} possuem réguas definadas nos valores 20% e 80%, respectivamente. Assim, espera-se que os valores de $Inc_{\text{sítio}}$ e Inc_{erro} estejam acima de 75% pois, nestas situações, eles indicam que ao aplicar o par (H,C) há uma probabilidade de 75% de o conjunto de requisitos de teste selecionados incluir informação útil para a depuração — o sítio do defeito ou um *rt-re*. Analogamente, desejam-se valores acima de 80% para Ef_{erro}, o que implica que 80% dos requisitos de teste selecionados são reveladores de erros. Já os valores interessantes para Ef_{cmd} são aqueles abaixo de 20%, pois indicam que o trecho de código selecionado é apenas 20% do trecho obtido utilizando H1. Os valores de corte foram estabelecidos arbitrariamente, mas definem uma relação de inclusão-eficiência satisfatória.

Os valores $Cmd_{\text{médio}}$ e $CE_{\text{médio}}$ selecionados pelos pares (H,C) nos cenários TP-C, GT-C e MT-C são apresentados, respectivamente, nas Tabelas 4.3, 4.4 e 4.5. Considerando-se os resultados relativos ao cenário TP-C descritos na Tabela 4.3, pode-se observar que a aplicação bem sucedida de (H3, todos ramos) em média selecionou 24 comandos (excluem-se declarações de variáveis) e 10,27 ramos suspeitos para a versão defeituosa número 1 do programa `Sort`.

Note-se que tanto nos gráficos contidos nas Figuras 4.3 a 4.20 como nas Tabelas 4.3, 4.4 e 4.5 os

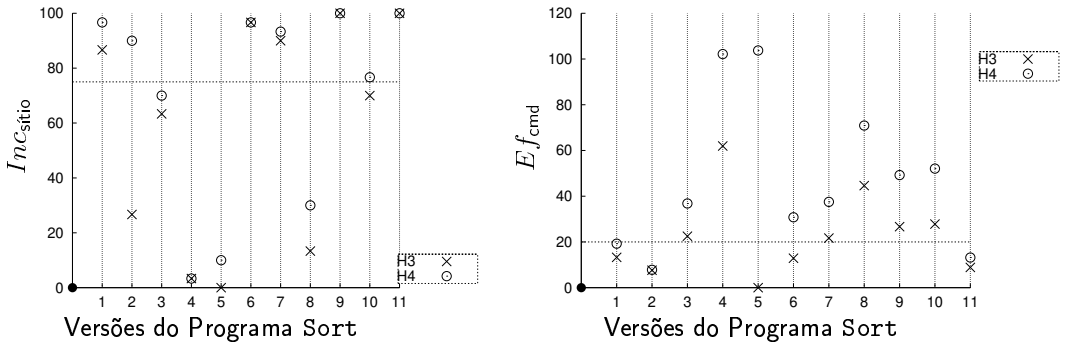
dados apresentados estão restritos às heurísticas do tipo *ranking*. A razão para a omissão das heurísticas H1 e H2 é que elas apresentaram um comportamento quase uniforme em todas as versões defeituosas, para todos os cenários, utilizando qualquer requisito de teste. Na verdade, o desempenho de H1 é alterado apenas no cenário GT-C pois, tanto no cenário TP-C como MT-C, existe apenas um caso de teste revelador de defeito. Quanto a H2, o cenário não altera o desempenho dessa heurística porque apenas dois casos de teste são sempre utilizados. O pares (H1, critério de teste) e (H2, critério de teste) obtiveram excelentes taxas de inclusão (próximas de 100% tanto em termos de $Inc_{\text{sítio}}$ como de Inc_{erro}), porém, com baixa eficiência (i.e., Ef_{cmd} alta e Ef_{erro} baixa). Em geral, essas heurísticas selecionaram um número muito grande de requisitos e, conseqüentemente, de comandos a serem examinados. Neste sentido, essas heurísticas continuam apresentando o mesmo problema das técnicas de *fatiamento*, qual seja, o grande número de comandos a serem investigados. Por isso, as heurísticas viáveis para aplicações reais são as heurísticas H3 e H4. Os dados apresentados neste capítulo restringem-se a elas para facilitar a visualização e análise. Os dados completos relativos a todas as heurísticas, requisitos de teste e cenários estão no Apêndice B.

A Tabela 4.6 contém os dados relativos à comparação de custo. Os dados referem-se apenas aos dados da versão 4. Os dados das demais versões não são apresentados porque são muito similares aos da versão 4 visto que o número de requisitos de teste exigidos e o tamanho dos conjuntos de casos de teste são pouco alterados de uma versão defeituosa para outra. A Tabela 4.6 apresenta, para cada critério, o número de requisitos exigidos ($|RT_C|$) e, para cada critério e cenário de teste-depuração, o tamanho médio dos conjuntos de casos de teste (média($|T_i|$)) e o custo empírico Custo$_{(H,C)}$.

Para facilitar a comparação, foi adotado que o custo de aplicação dos pares (heurística, todos ramos) é igual a 1 para todos os cenários, de forma que os custos dos pares (heurística, todos usos) e (heurística, todos potenciais usos) representam o incremento em relação aos custos dos pares (heurística, todos ramos). Deste modo, considerando o critério todos potenciais usos (linha 3 da Tabela 4.6), o número de requisitos exigidos é 5263, os conjuntos T_i, em média, são compostos por 25,27, 75,67 e 71,00 casos de teste, respectivamente, nos cenários TP-C, GT-C e MT-C e os custos de (heurística, todos potenciais usos) são 40,77, 50,23 e 46,94 vezes maiores que os custos de (heurística, todos ramos), respectivamente, para os cenários TP-C, GT-C e MT-C.

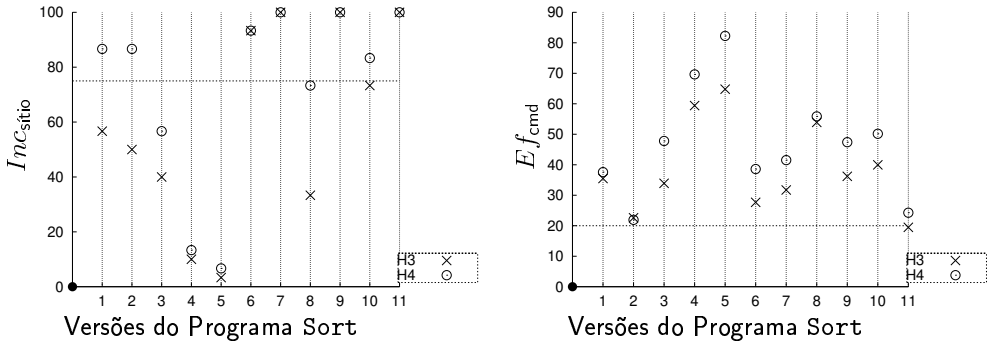
4.5.2 Resultados

A discussão dos resultados a seguir é organizada em função do cenário utilizado. No cenário TP-C, considerando-se as medidas $Inc_{\text{sítio}}$ e Ef_{cmd}, observa-se que apesar de os pares (H4, critério de teste) serem inclusivos a eficiência deixou a desejar. Nos gráficos das Figuras 4.3 a 4.5, nota-se que a heurística H4, utilizando todos ramos e todos potenciais usos, obteve valores de $Inc_{\text{sítio}}$ maiores ou próximos de 75% para oito versões defeituosas; porém, em geral, ela seleciona um conjunto de comandos com mais de 20% dos comandos selecionados por H1.



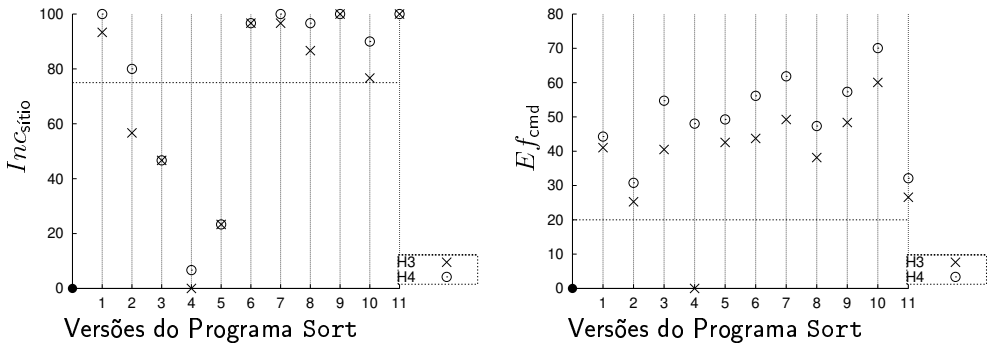
(a) (b)

Figura 4.3: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos ramos) no cenário TP-C.



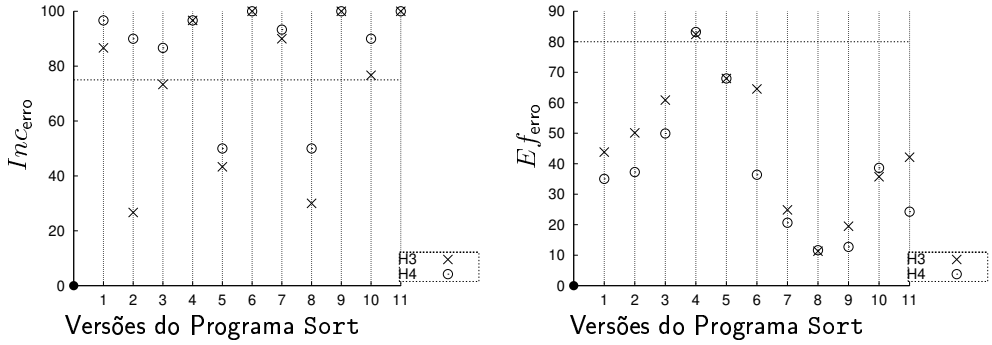
(a) (b)

Figura 4.4: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos usos) no cenário TP-C.



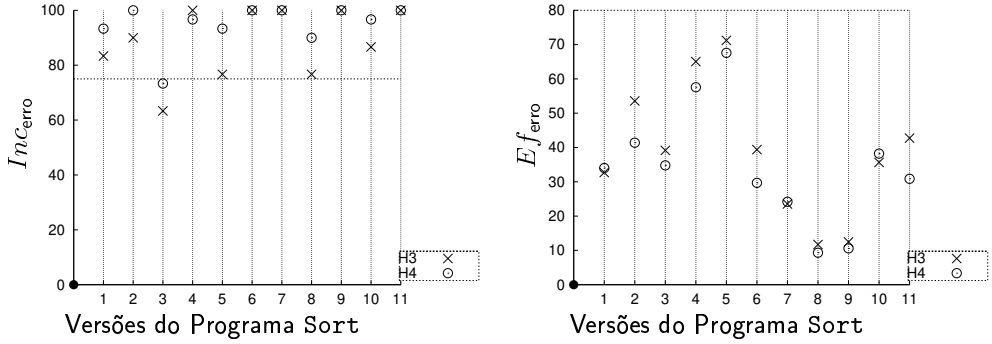
(a) (b)

Figura 4.5: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos potenciais usos) no cenário TP-C.



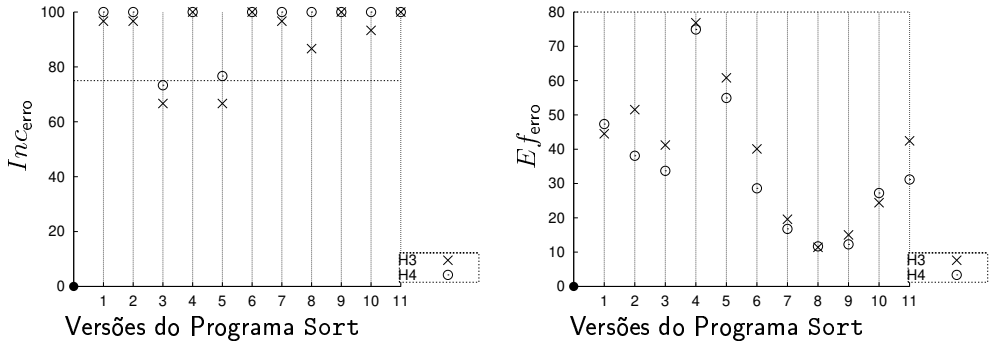
(a) (b)

Figura 4.6: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos ramos) no cenário TP-C.



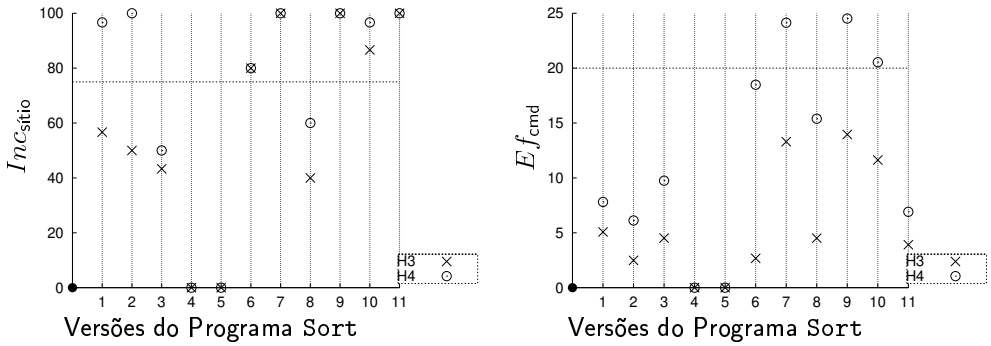
(a) (b)

Figura 4.7: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos usos) no cenário TP-C.



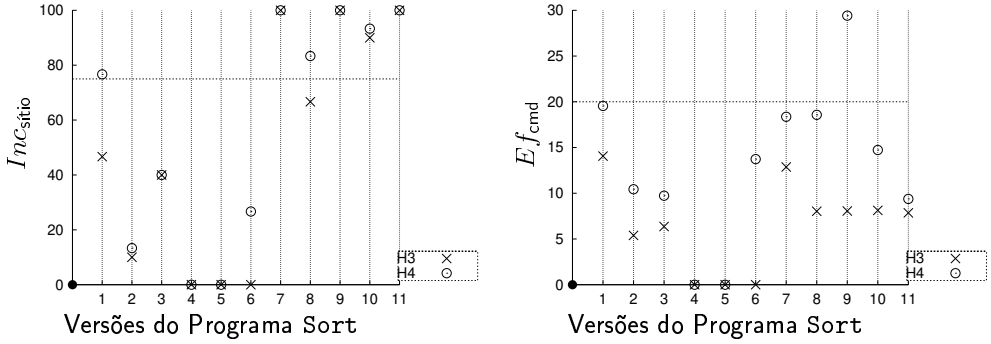
(a) (b)

Figura 4.8: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos potenciais usos) no cenário TP-C.



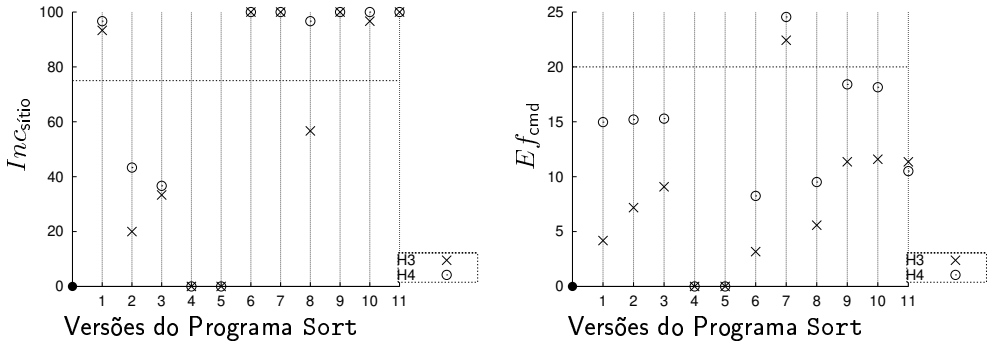
(a) (b)

Figura 4.9: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos ramos) no cenário GT-C.



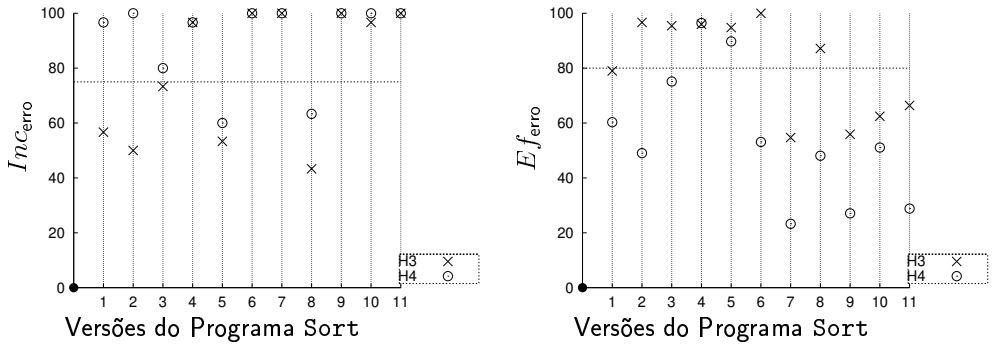
(a) (b)

Figura 4.10: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos usos) no cenário GT-C.



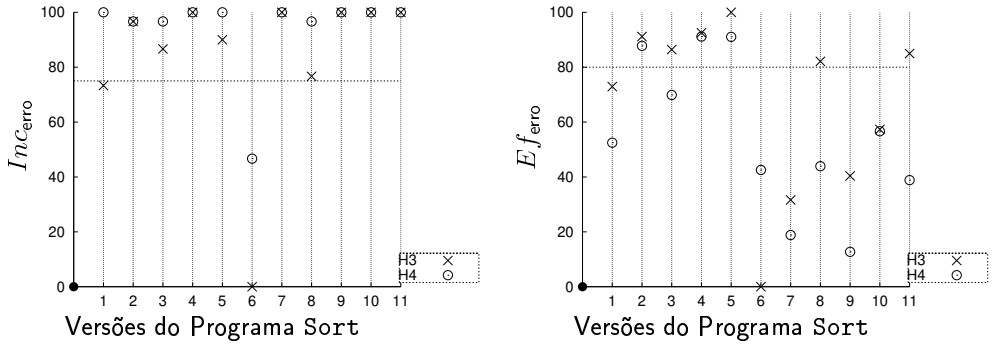
(a) (b)

Figura 4.11: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos potenciais usos) no cenário GT-C.



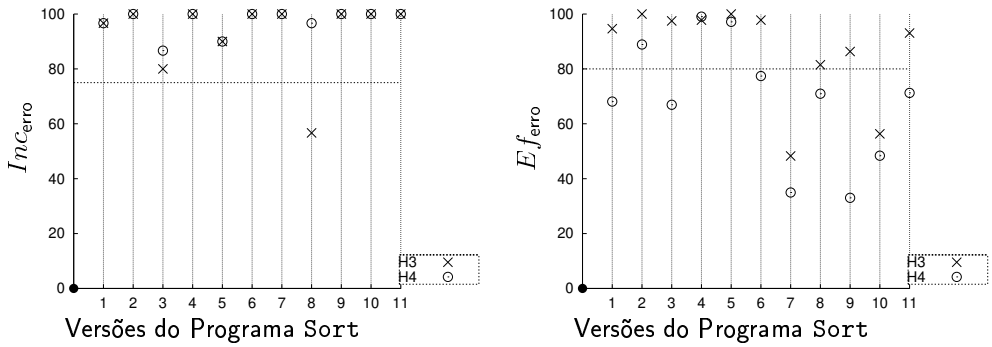
(a) (b)

Figura 4.12: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos ramos) no cenário GT-C.



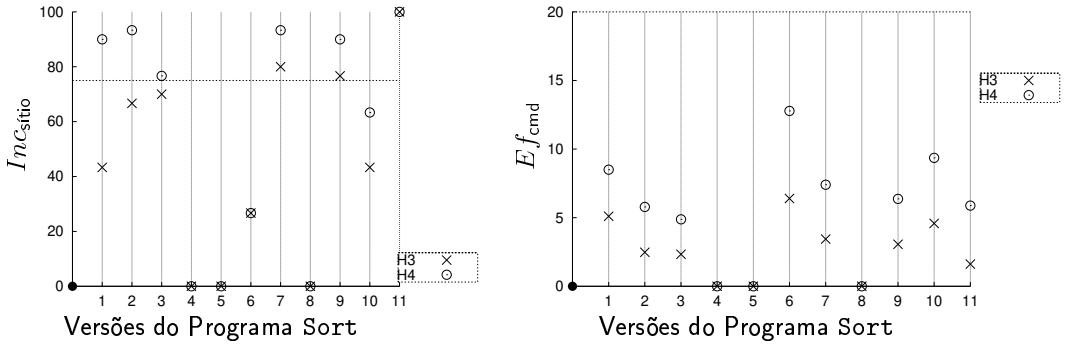
(a) (b)

Figura 4.13: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos usos) no cenário GT-C.



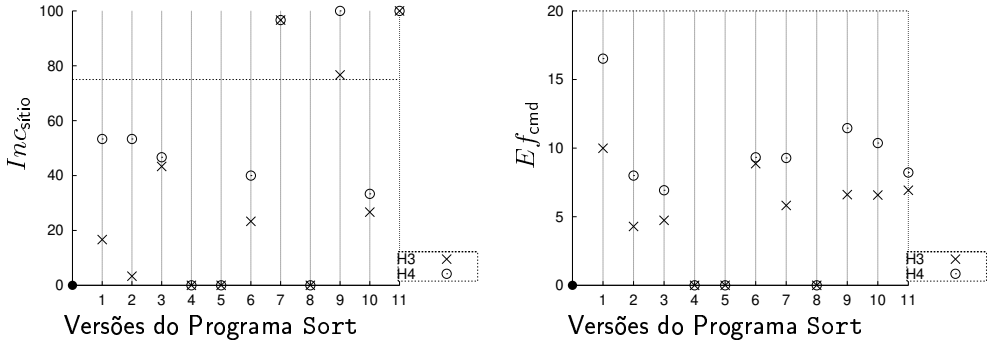
(a) (b)

Figura 4.14: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos potenciais usos) no cenário GT-C.



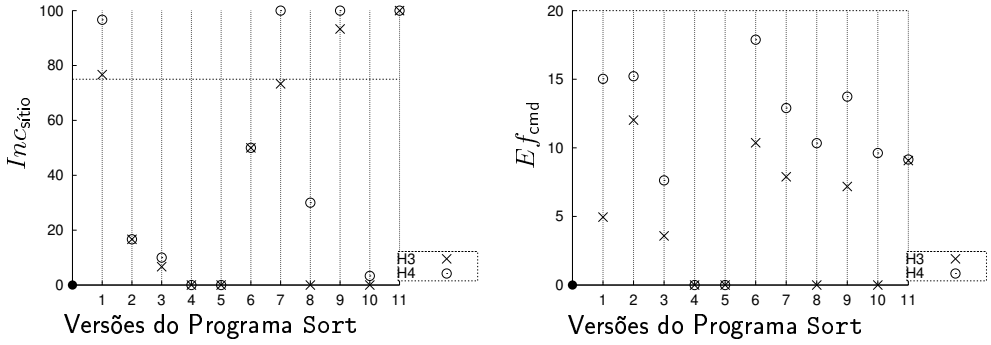
(a) (b)

Figura 4.15: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos ramos) no cenário MT-C.



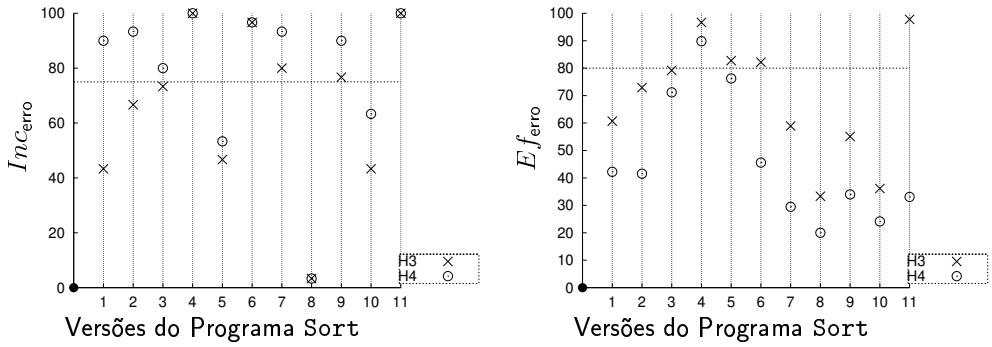
(a) (b)

Figura 4.16: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos usos) no cenário MT-C.



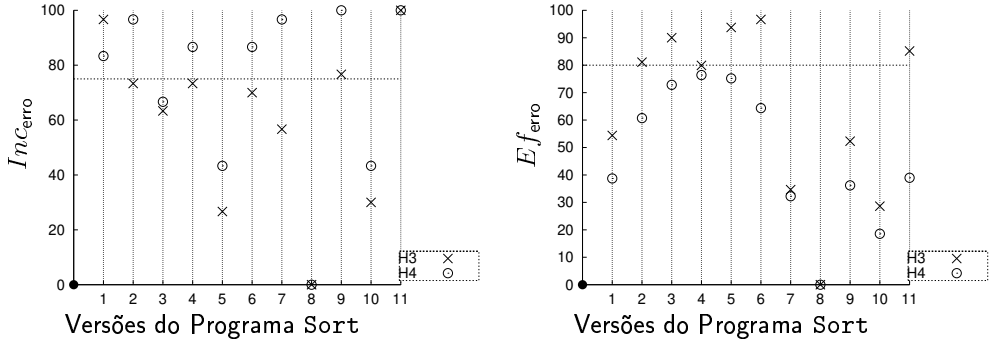
(a) (b)

Figura 4.17: Medidas $Inc_{\text{sítio}}$ e Ef_{cmd} para os pares (heurística, todos potenciais usos) no cenário MT-C.



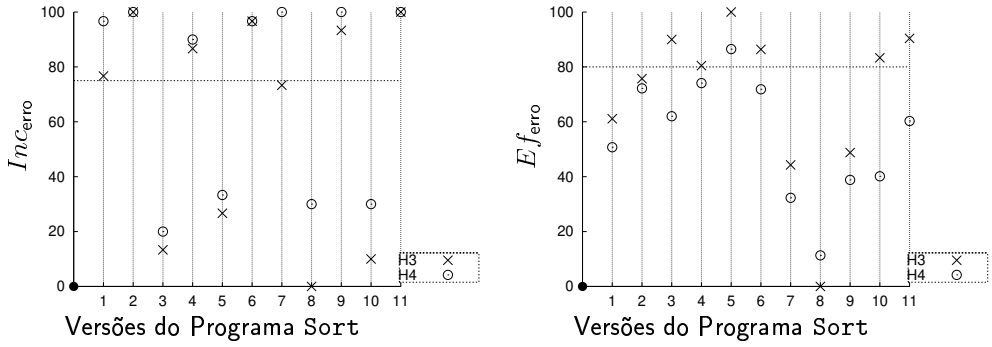
(a) (b)

Figura 4.18: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos ramos) no cenário MT-C.



(a) (b)

Figura 4.19: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos usos) no cenário MT-C.



(a) (b)

Figura 4.20: Medidas Inc_{erro} e Ef_{erro} para os pares (heurística, todos potenciais usos) no cenário MT-C.

Tabela 4.3: Número médio de comandos ($Cmd_{\text{médio}}$) e requisitos de teste ($CE_{\text{médio}}$) selecionados no cenário TP-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Ramos	H3	$Cmd_{\text{médio}}$	24,00	11,75	30,58	57,00	0,00	18,97	32,52	48,25	41,30	43,48	12,30
		$CE_{\text{médio}}$	10,27	5,50	12,32	5,90	18,85	7,40	13,93	20,78	18,70	16,30	4,40
	H4	$Cmd_{\text{médio}}$	34,93	11,52	48,81	94,00	175,33	45,03	56,39	88,44	76,57	82,78	18,23
		$CE_{\text{médio}}$	15,83	5,70	19,27	10,93	29,27	15,73	23,29	29,60	31,93	28,22	7,13
Todos Usos	H3	$Cmd_{\text{médio}}$	98,35	51,73	80,75	128,00	173,00	66,86	74,47	120,90	91,27	104,91	44,27
		$CE_{\text{médio}}$	123,52	43,04	128,89	67,20	109,09	77,90	133,60	124,78	167,13	236,58	84,80
	H4	$Cmd_{\text{médio}}$	105,08	49,96	111,41	151,00	194,50	92,39	96,87	121,68	118,43	132,40	55,13
		$CE_{\text{médio}}$	190,75	68,53	224,18	107,90	161,86	161,97	212,57	226,81	284,70	325,72	119,23
Todos Potenciais Usos	H3	$Cmd_{\text{médio}}$	114,50	57,94	97,79	0,00	121,00	99,21	119,17	79,96	118,47	152,00	59,03
		$CE_{\text{médio}}$	210,17	50,62	126,80	48,10	113,15	130,57	253,62	110,15	300,97	268,54	72,40
	H4	$Cmd_{\text{médio}}$	123,80	70,83	132,36	101,50	138,71	128,79	149,73	99,03	140,33	178,11	71,33
		$CE_{\text{médio}}$	280,67	92,73	220,77	69,37	173,57	258,07	387,13	189,03	431,47	398,53	102,50

Tabela 4.4: Número médio de comandos ($Cmd_{\text{médio}}$) e requisitos de teste ($CE_{\text{médio}}$) selecionados no cenário GT-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Ramos	H3	$Cmd_{\text{médio}}$	7,06	2,13	2,08	0,00	0,00	1,58	6,27	1,83	6,67	9,12	5,07
		$CE_{\text{médio}}$	4,41	1,07	1,68	2,38	2,44	1,23	2,67	2,31	3,13	3,31	2,03
	H4	$Cmd_{\text{médio}}$	11,86	6,27	4,80	0,00	0,00	10,42	11,23	6,78	11,37	16,83	8,70
		$CE_{\text{médio}}$	6,66	3,23	3,75	4,83	4,78	3,00	4,87	4,26	5,30	6,60	3,60
Todos Usos	H3	$Cmd_{\text{médio}}$	27,00	7,00	7,75	0,00	0,00	0,00	14,17	10,25	8,93	10,93	16,13
		$CE_{\text{médio}}$	19,05	2,62	4,19	7,93	2,48	0,00	13,97	9,43	5,90	6,67	27,07
	H4	$Cmd_{\text{médio}}$	38,74	13,00	11,83	0,00	0,00	15,12	20,33	23,80	32,80	20,04	19,27
		$CE_{\text{médio}}$	39,07	5,52	9,41	15,80	6,23	6,07	22,83	23,72	96,70	20,20	59,20
Todos Potenciais Usos	H3	$Cmd_{\text{médio}}$	8,71	8,00	8,00	0,00	0,00	3,53	22,20	6,76	10,73	11,79	22,23
		$CE_{\text{médio}}$	2,90	3,53	2,83	10,87	1,89	2,43	29,37	2,94	7,67	6,50	21,63
	H4	$Cmd_{\text{médio}}$	30,45	17,85	13,64	0,00	0,00	9,17	24,30	11,48	17,47	18,73	20,37
		$CE_{\text{médio}}$	34,31	7,47	6,04	18,20	7,56	6,03	40,80	5,83	21,37	11,60	30,83

Tabela 4.5: Número Médio de comandos ($Cmd_{\text{médio}}$) e requisitos de teste ($CE_{\text{médio}}$) selecionados no cenário MT-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Ramos	H3	$Cmd_{\text{médio}}$	8,54	3,45	3,24	0,00	0,00	10,50	5,04	0,00	4,61	7,00	2,20
		$CE_{\text{médio}}$	5,54	1,65	2,00	2,27	2,36	2,00	2,79	3,00	3,35	3,46	1,07
	H4	$Cmd_{\text{médio}}$	14,67	8,21	6,61	0,00	0,00	18,88	10,64	0,00	9,70	14,53	8,03
		$CE_{\text{médio}}$	8,41	3,86	3,83	4,50	4,94	3,86	5,75	5,00	6,00	7,21	3,03
Todos Usos	H3	$Cmd_{\text{médio}}$	26,80	9,00	11,62	0,00	0,00	20,43	13,03	0,00	16,00	17,00	16,00
		$CE_{\text{médio}}$	16,12	3,59	4,53	3,09	2,50	5,00	12,65	0,00	9,35	8,22	27,00
	H4	$Cmd_{\text{médio}}$	45,25	18,44	17,00	0,00	0,00	21,25	21,07	0,00	27,83	26,50	19,00
		$CE_{\text{médio}}$	51,32	10,76	12,75	11,96	6,85	6,77	20,59	0,00	26,53	17,38	59,00
Todos Potenciais Usos	H3	$Cmd_{\text{médio}}$	14,04	32,80	8,50	0,00	0,00	25,87	18,27	0,00	17,46	0,00	20,27
		$CE_{\text{médio}}$	9,91	4,30	2,75	4,31	5,50	11,72	22,32	0,00	21,46	2,00	22,43
	H4	$Cmd_{\text{médio}}$	42,62	41,60	20,67	0,00	0,00	44,20	30,37	22,11	33,57	25,00	20,43
		$CE_{\text{médio}}$	56,59	9,20	10,17	16,85	10,90	21,79	51,17	16,33	54,80	8,00	36,30

Tabela 4.6: Comparação do custo empírico de aplicação dos pares (heurística, todos ramos), (heurística, todos usos) e (heurística, todos potenciais usos) em diferentes cenários

Critério	$\lvert RT_C \rvert$	Média($\lvert T_i \rvert$) para			Custo$_{(H,C)}$ para		
		TP-C	GT-C	MT-C	TP-C	GT-C	MT-C
Todos Ramos	249	13,10	31,84	31,97	1	1	1
Todos Usos	2398	20,77	53,93	53,37	15,27	16,31	16,07
Todos Potenciais Usos	5263	25,27	75,67	71,00	40,77	50,23	46,94

Com relação às medidas em termos de *rt-re*s, as heurísticas H3 e H4 obtiveram valores de Inc_{erro} acima de 75% para a maioria dos programas do estudo de caso, principalmente quando foram utilizados requisitos de critérios de fluxo de dados. No entanto, o número de requisitos suspeitos selecionados é muito grande (veja Tabela 4.3), o que acarretou medidas de eficiência Ef_{erro} muito aquém do desejado (80%).

No cenário GT-C, os pares (heurística tipo *ranking*, critério de teste) apresentaram os melhores resultados. Com relação à localização do sítio do defeito, os resultados são razoáveis (sete versões obtiveram $Inc_{\text{sítio}} > 75\%$ para H4 utilizando todos ramos e todos potenciais usos) com uma redução expressiva do espaço de busca (a medida Ef_{cmd} foi menor que 20% para a maioria das versões e, em termos absolutos, o número de comandos identificado foi bastante reduzido).

Contudo, os melhores resultados no cenário GT-C são obtidos quando são considerados *rt-re*s. Neste contexto, os pares (H3, requisitos de teste de fluxo de dados) são muito inclusivos (10 versões obtêm Inc_{erro} maior ou próximo de 75% para todos usos e todos potenciais usos); e, em termos de eficiência, os resultados são satisfatórios porque para a maioria das versões Ef_{erro} está acima de 80% e, mesmo quando isto não ocorre, o número de requisitos de teste selecionados é pequeno (veja Tabela 4.4), o que facilita a identificação dos requisitos que revelam erros. Nessa mesma linha, os resultados de (H4, todos ramos) são também satisfatórios, pois Inc_{erro} é maior que 75% para 9 versões e o número de ramos selecionados é reduzido, mesmo a medida Ef_{erro} não sendo a desejada.

No cenário MT-C, as heurísticas H3 e H4 não apresentaram habilidade para localizar o defeito quando foram utilizados requisitos de teste de fluxo de dados. Quando o par (H4, todos ramos) foi aplicado os resultados foram apenas razoáveis, pois somente seis versões obtiveram $Inc_{\text{sítio}} > 75\%$ e $Ef_{\text{cmd}} < 20\%$. Quando *rt-re*s foram considerados, os pares (H3, requisitos de teste de fluxo de dados) passaram também a apresentar resultados razoáveis. Por exemplo, para o par (H3, todos potenciais usos), sete versões obtiveram $Inc_{\text{erro}} > 75\%$, sendo que três delas com $Ef_{\text{erro}} > 80\%$. Porém, analogamente ao observado no cenário GT-C, nas versões em que o valor da medida Ef_{erro} não é o desejado, o número de requisitos suspeitos selecionado é pequeno.

Uma observação importante do experimento é que os pares (heurísticas tipo *ranking*, critério de teste) foram mais *inclusivos* e *eficientes* na seleção de *rt-re*s do que na seleção de um trecho de código que inclui o defeito. Isto ocorre porque eles apresentam habilidade para selecionar *rt-re*s mesmo quando a probabilidade de acertar o sítio do defeito é baixa (como ocorre nas versões 2 e 3 no cenário GT-C) ou mesmo nula (caso das versões 4 e 5 no cenário GT-C). Portanto, ao utilizar os pares (heurística tipo *ranking*, critério de teste) na depuração, o mantenedor deve esperar muitas vezes obter somente indicações úteis para localização do defeito e não o sítio do defeito.

4.5.3 Ameaças à Validade

O estudo de caso realizado procura simular a aplicação dos pares (heurística, critério de teste) em programas defeituosos dentro de cenários de teste-depuração. No entanto, deve-se ressaltar que os programas, os conjuntos de casos de teste nos quais são aplicados os pares (heurística, critério de teste) e os cenários de teste-depuração são aproximações da realidade. Por isso, os resultados devem ser analisados tendo em vista as limitações dessas aproximações.

A primeira limitação é que o experimento utiliza diferentes versões de um único programa. Outra limitação é que as versões do programa `Sort` possuem um único defeito; na prática vários defeitos estão presentes no programa.

Entretanto, dependendo de como os pares (heurística, critério de teste) são aplicados, a simplificação representada pela presença de um único defeito não se distancia completamente do que ocorre na prática. Suponha-se que as falhas observadas tenham sido classificadas e os pares (heurísticas,critério de teste) aplicados em conjuntos nos quais os casos de teste que revelam defeitos provocam a ocorrência de apenas um tipo de falha. Este procedimento tende naturalmente a ser utilizado visto que se baseia na intuição de que falhas distintas são causadas por defeitos distintos. Estudos teóricos recentes demostram que a possibilidade de acoplamento dos defeitos é reduzida[4] (Wah, 2000), o que indica que esta intuição pode estar correta.

Assim, considerando-se que os pares (heurísticas,critério de teste) são aplicados nas situações descritas acima e que cada defeito tende a provocar somente um tipo de falha, então a presença de um único defeito nas versões defeituosas do `Sort` pode ser considerada uma aproximação inicial razoável.

Com relação à aplicação dos pares (heurística, critério de teste), os resultados foram obtidos da aplicação direta das heurísticas sem levar em consideração conhecimento prévio do mantenedor. No entanto, o desempenho dos pares (heurística, critério de teste) pode ser diferente se o conjunto de casos de teste no qual os pares são aplicados for selecionado utilizando conhecimento prévio do mantenedor. Por exemplo, a heurística H2 poderá ter o seu desempenho melhorado se o mantenedor for capaz de selecionar casos de teste t (revelador de defeito) e t' (não-revelador de defeito) com saídas muito semelhantes, de forma que um grande número de requisitos de teste comuns é eliminado na operação de subtração da heurística. Abordagens desse tipo têm sido sugeridas para a compreensão de programas (Wong et al., 1999; Agrawal et al., 1998).

Os cenários de teste-depuração, por sua vez, representam situações de *pior caso*. Por exemplo, no cenário TP-C é assumido que todos os conjuntos são não-adequados. Essa é a característica esperada da maioria dos conjuntos selecionados nesse cenário. Contudo, nada impede que o caso de teste revelador de defeito (o último selecionado no cenário TP-C) exercite um subconjunto de requisitos que completa a cobertura do critério, tornando o conjunto selecionado adequado. Este conjunto em particular terá as

[4]Segundo Wah (2000), "o acoplamento de defeitos é o fenômeno em que um conjunto de casos de teste detecta defeitos quando ocorrem isoladamente no programa, mas não os detecta quando ocorrem ao mesmo tempo".

mesmas características de um conjunto do cenário GT-C. Da mesma forma, no cenário MT-C assume-se que apenas um caso de teste revelador de defeito foi identificado depois da liberação do software; porém, pode ocorrer de mais de um deles ser identificado em campo. Portanto, adverte-se que os resultados nos cenários TP-C e MT-C podem ser subestimados.

Concluindo, cabe neste estudo de caso a mesma ressalva feita no estudo do Capítulo 3, pois os resultados são igualmente influenciados por fatores como o tipo do defeito, o domínio de aplicação e a estrutura do programa, bem como a arquitetura do sistema.

4.6 Discussão

4.6.1 Influência do Cenário

O excepcional desempenho das heurísticas do tipo *ranking* para o cenário GT-C é explicado pela sua habilidade em se beneficiar da atividade de teste sistemática na qual são desenvolvidos conjuntos adequados de casos de teste. Elas se concentram nos requisitos exercitados pelos casos de teste reveladores de defeito, filtrando, por meio do seu mecanismo de classificação, aqueles que têm pouca correlação com a ocorrência da falha.

Para o cenário MT-C, o mesmo efeito descrito acima ocorre, porém, não de forma tão evidente devido às características dos conjuntos utilizados. Eles são adequados ou quase-adequados e contêm apenas um caso de teste revelador de defeito. Mesmo assim, as heurísticas H3 e H4 são ainda capazes de identificar informação útil (*rt-re*s) para a depuração, beneficiando-se do fato de a maioria dos requisitos ter sido exercitada por casos de teste não-reveladores de defeito. Portanto, a alta cobertura em relação critério de teste também contribui para a identificação de *rt-re*s.

Os resultados do cenário TP-C, por sua vez, são esperados pois os conjuntos de casos de teste neste cenário não possuem as duas características que explicam o bom desempenho dos dois anteriores. No cenário TP-C, os conjuntos *não* são adequados e possuem apenas *um* caso de teste revelador de defeito; por isso, o peso associado aos requisitos não é suficiente para estabelecer uma classificação que elimine os requisitos menos relacionados com o defeito.

Contudo, do ponto de vista prático, o cenário TP-C é o mais realista visto que a tendência natural do testador, ao detectar a presença de um defeito, é solicitar a depuração do programa antes de retomar o teste. Neste sentido, o cenário GT-C é o menos comum na prática pois requer a continuação do teste mesmo depois da ocorrência de uma falha. Já o cenário MT-C pode ser considerado realista porque o uso de informação de teste na depuração pressupõe a realização do teste estrutural sistemático e o armazenamento da informação gerada.

É importante ressaltar ainda que no cenário MT-C o processo de depuração é muitas vezes conduzido por um mantenedor com pouco ou nenhum conhecimento do software; portanto, o fato do número de requisitos suspeitos selecionados ser pequeno e possuir uma probabilidade razoável de conter *rt-re*s

constitui um ponto de partida para onde direcionar as atenções do mantenedor.

Dessa maneira, os resultados obtidos indicam que, para se obter o máximo benefício da informação de teste estrutural na depuração, é necessário aplicar as heurísticas em conjuntos de casos de teste que estejam próximos daqueles desenvolvidos nos cenários GT-C e MT-C. Estes conjuntos podem ser obtidos criando-se mais casos de teste que possuam uma das seguintes características: sejam reveladores de defeito ou, caso não o sejam, aumentem a cobertura em relação ao critério de teste.

4.6.2 Desempenho dos Requisitos de Teste *versus* Custo

Todos os requisitos de teste (ramos, *adu*s e *adpu*s) apresentaram habilidade para localização de defeitos. As *adpu*s apresentaram as melhores medidas de inclusão e eficiência usando as heurísticas do tipo *ranking*. Seu custo, porém, ultrapassou demasiadamente os custos associados aos outros tipos de requisitos. Na Tabela 4.6 pode-se observar que a utilização de *adpu*s chega a custar 50 vezes mais do que a utilização de ramos. As *adu*s, por sua vez, custam por volta de 16 vezes mais.

Embora os resultados dos pares (heurística tipo *ranking*, todos potenciais usos) para os programas do estudo de caso sejam melhores que os resultados dos pares (heurística tipo *ranking*, todos ramos), eles *não* são *marcadamente superiores*, especialmente quando a heurística H4 é utilizada. Neste sentido, pode-se argumentar que os requisitos dos critérios todos ramos possuem a melhor relação custo-benefício.

Entretanto, duas situações devem ser levadas em consideração. Na primeira, o testador já realizou o teste de fluxo de dados; portanto, o custo extra já foi amortizado. Ainda assim, uma estratégia incremental pode ser utilizada. O mantenedor pode começar coletando os resultados da aplicação dos pares (heurística tipo *ranking*, todos ramos) para procurar pelo sítio do defeito visto que em geral o pedaço de código selecionado é menor. Se a busca pelo sítio do defeito não for bem sucedida, ele pode então utilizar os pares (heurística tipo *ranking*, critérios de fluxo de dados). A idéia é não somente procurar pelo sítio do defeito no trecho selecionado por estes pares, mas também investigar as *adu*s e *adpu*s suspeitas quanto a sua capacidade de revelar erros. Isto porque os requisitos dos critérios de fluxo de dados tendem a fornecer mais indicações úteis para a localização do defeito pois envolvem uma definição e um (potencial) uso, o que facilita a identificação de *rt-re*s e, conseqüentemente, do defeito.

Na segunda situação, o testador aplicou apenas o teste de ramos e não pretende aplicar o teste de fluxo de dados. Neste caso, o mantenedor pode repetir o primeiro passo acima e verificar se sítio do defeito foi incluído. Caso não tenha sido, ele pode realizar a análise de adequação de alguns casos de teste reveladores de defeito com relação aos critérios de fluxo de dados e investigar as *adu*s e *adpu*s vinculadas ao ramos selecionados pelo par (heurística do tipo *ranking*, todos ramos).

4.6.3 Idiossincrasia do Defeito

A idiossincrasia do defeito tem um papel importante nos resultados obtidos pelos pares (heurística, critério de teste) visto que a simples localização do defeito pode inviabilizar a habilidade de um requisito

de teste em fornecer informação útil para a depuração.

Para exemplificar a influência da idiossincrasia do defeito, considere-se a situação que acontece quando os pares (heurística, todos usos) são utilizados no cenário GT-C (veja Tabela B.5) para localizar o defeito da versão número 6. Neste caso, o defeito está localizado no teste de um laço e as falhas ocorrem tanto devido a casos de teste que *entram* no corpo do laço quanto a casos de teste que *não* entram. O sítio do defeito é atingido em ambos os casos, mas as *adu*s exercitadas são diferentes.

Por exemplo, usando a heurística H1, que implementa uma operação de intersecção de todas as *adu*s exercitadas pelos casos de teste reveladores de defeito, a medida $Inc_{\text{sítio}}$ é de apenas 6,67%. Isto ocorre porque a probabilidade de selecionar conjuntos T_i no quais há um subconjunto de *adu*s exercitadas que alcançam o sítio de defeito e que são exercitadas por *todos* os casos de teste reveladores de defeito de T_i é baixa. Na maioria das vezes, o que acontece é que as *adu*s que alcançam o sítio de defeito são eliminadas na operação de intersecção de conjuntos de H1. A heurística H3 usando *adu*s é prejudicada da mesma maneira, pois o sítio do defeito não é atingido e nenhum *rt-re* consegue peso suficiente para ser selecionado. Por outro lado, as mesmas heurísticas utilizando ramos e *adpu*s não são prejudicadas pela idiossincrasia do defeito.

Em outras situações, a habilidade de acertar o sítio do defeito é prejudicada, mas não a de selecionar *rt-re*s. Considere-se a versão 3 na qual o defeito está localizado na definição de uma instrução macro. Este fato faz com que as falhas ocorram quando são executados casos de teste que atingem diferentes pontos do programa. As heurísticas do tipo *ranking*, no seu melhor cenário (GT-C), não obtêm um bom desempenho em termos de inclusão do sítio do defeito; porém, em termos de *rt-re*s, o desempenho é bem razoável. Já nas versões 4 e 5, os defeitos afetam os valores de variáveis globais de forma que seus efeitos somente são sentidos mais *tarde* na execução do programa e em outra função. Em termos de inclusão do sítio do defeito, as heurísticas do tipo *ranking* falham completamente, porém, quando *rt-re*s são considerados, elas são muito inclusivas e eficientes.

Portanto, os dados indicam que a identificação de *rt-re*s ajuda a reduzir a influência da idiossincrasia do defeito, bem como a utilização de requisitos de teste de diferentes critérios. Neste sentido, esta última indicação corrobora a adoção da estratégia incremental discutida na seção anterior.

4.6.4 Experimentos Semelhantes

Collofello e Cousins (1987) conduziram um experimento utilizando um conjunto pequeno de programas escritos em Pascal em que heurísticas do tipo *fatiamento*, *recorte* e *ranking* foram aplicadas utilizando os requisitos de teste do critério todos nós. Os defeitos foram semeados nos programas e um único conjunto de casos de teste foi desenvolvido para cada um deles. Os resultados indicaram que as heurísticas equivalentes a H2 e H3 obtiveram as melhores medidas em termos de inclusão e eficiência.

Pan e Spafford (1992) realizaram um experimento baseado em programas pequenos escritos em linguagem C. Diferentes heurísticas utilizando *fatias* dinâmicas de programas foram aplicadas. As *fatias*

foram obtidas utilizando casos de teste de um conjunto adequado ao critério todos usos e como critério de *fatiamento* as variáveis de saída incorretas no último comando executado. Novamente, as heurísticas do tipo *ranking* obtiveram os melhores resultados.

Agrawal et al. (1995) apresentam um experimento em que o par (H2, todos nós) foi aplicado em várias versões do programa `Sort` utilizando um único conjunto de casos de teste adequado ao critério todos usos. Os autores observaram que o par (H2, todos nós) é inclusivo mas não suficientemente eficiente.

O estudo de caso realizado analisa o uso de informação de teste na depuração por meio de heurísticas em diferentes perspectivas. Ele utiliza diferentes requisitos de teste e vários conjuntos de casos de teste desenvolvidos considerando três cenários de teste-depuração. Além disso, aspectos de custo são levados em consideração. Porém, o aspecto mais importante do experimento é a utilização de *requisitos de teste reveladores de erros* como medida de avaliação da habilidade dos pares (H,C) em fornecer informação útil para a depuração.

4.7 Considerações Finais

Neste capítulo, foi apresentada uma análise empírica do uso de informação de teste estrutural na depuração por meio de heurísticas. O objetivo foi analisar a habilidade dos pares (heurística, critério de teste) em fornecer informação útil para a depuração. Essa informação útil pode estar na forma de um trecho de código onde o defeito está localizado ou um conjunto de requisitos de teste que fornecem indicações em tempo de execução que podem levar o mantenedor a localizar o defeito.

Para modelar esse tipo de requisito, foram introduzidos os *requisitos de teste reveladores de erro* (*rt-re*). Esses requisitos, quando analisados em tempo de execução, apresentam discrepâncias em relação ao comportamento esperado ou desvios em relação à especificação (erros) que constituem indicações úteis à depuração.

A análise empírica conduzida avaliou aspectos de inclusão, eficiência e custo. As medidas de inclusão têm como objetivo avaliar a habilidade do par (heurística, critério de teste) em incluir o defeito no trecho de código selecionado ou identificar *rt-re*s. As medidas de eficiência visam medir o tamanho do espaço de busca identificado. A medida de custo determina o custo empírico de se utilizar os requisitos de teste de um particular critério para localizar o defeito utilizando heurísticas.

As medidas de inclusão, eficiência e custo foram obtidas a partir dos resultados da aplicação de heurísticas de diferentes tipos (*fatiamento*, *recorte* e *ranking*) utilizando requisitos de três critérios de teste (todos ramos, todos usos e todos potenciais usos) para localizar defeitos em versões incorretas do programa `Sort`. Os pares (heurística, critério de teste) foram aplicados em conjuntos de casos de teste desenvolvidos considerando diferentes cenários de teste-depuração (TP-C, GT-C e MT-C).

Pela análise dos dados, observou-se que o cenário, o requisito de teste em si e a idiossincrasia do defeito têm um papel importante na habilidade do par (heurística, critério de teste) em fornecer

informação útil para a depuração. Mesmo assim, algumas conclusões de caráter geral puderam ser obtidas.

Confirmando estudos anteriores, as heurísticas do tipo *fatiamento* e *recorte* obtiveram resultados *insatisfatórios* utilizando qualquer requisito de teste em todos os cenários. As heurísticas do tipo *ranking*, por sua vez, apresentaram os resultados mais *inclusivos* e *efetivos*, especialmente em termos de *rt-re*s, nos cenários GT-C e MT-C. Infelizmente, no cenário mais realista (TP-C), o desempenho dessas heurísticas também deixou a desejar.

Como as heurísticas do tipo *fatiamento* e *recorte* não são alternativas viáveis, as heurísticas do tipo *ranking* são ainda as mais promissoras para uso em aplicações reais. Porém, o estudo de caso indica que o mantenedor deve aplicar os pares (heurística tipo *ranking*, critério de teste) em conjuntos de casos de teste que estejam o mais próximo possível daqueles desenvolvidos nos cenários mais favoráveis (GT-C e MT-C). Isto pode ser obtido gerando-se casos de teste adicionais que provocam a ocorrência da falha ou que aumentem a cobertura do conjunto de casos de teste.

Os pares (heurística tipo *ranking*, critérios de fluxo de dados) obtiveram os melhores resultados em termos de inclusão e eficiência. Porém, o custo de utilização das *adu*s e *adpu*s foi muito superior ao custo associado ao uso de ramos. Isto não é um problema se o teste de fluxo de dados foi realizado. Entretanto, na hipótese de somente o teste de ramos ter sido conduzido, as *adu*s e *adpu*s podem ainda ser utilizadas para apoiar a localização do defeito. Caso seja necessário, o mantenedor pode realizar a análise de adequação com respeito aos critérios de fluxo de dados de alguns casos de teste reveladores de defeito e utilizar as *adu*s e *adpu*s associadas aos ramos identificados utilizando os pares (heurística tipo *ranking*, todos ramos) como os requisitos de teste a serem investigados. Este procedimento baseia-se no fato de que os critérios de fluxo de dados tendem a fornecer mais indicações úteis para a localização do defeito por envolverem uma definição e um (potencial) uso, o que facilita a identificação de *rt-re*s e, conseqüentemente, do defeito. Além disso, ele possibilita a utilização dos mecanismos a serem introduzidos no próximo capítulo.

Outro fator importante no desempenho dos pares (heurística, critério de teste) é a idiossincrasia do defeito. Os dados do estudo de caso mostram que a habilidade em fornecer informação útil para a depuração dos requisitos de teste de um critério pode ser completamente inviabilizada dependendo, por exemplo, da posição do defeito no código fonte ou do tipo de variáveis que ele influencia. Entretanto, esses mesmos dados indicam que a influência da idiossincrasia do defeito é menor quando *rt-re*s são considerados e os requisitos de diferentes critérios de teste estão disponíveis na depuração.

A observação mais importante do estudo de caso, porém, foi que os pares (heurística tipo *ranking*, critério de teste) — os mais promissores para aplicações reais — são mais inclusivos e eficientes em termos de *rt-re*s. Isto ocorre porque eles apresentam habilidade para selecionar *rt-re*s mesmo quando a probabilidade de acertar o sítio do defeito é baixa ou mesmo nula. Portanto, muitas vezes, o mantenedor vai obter somente indicações úteis para localização do defeito, e não o sítio do defeito, a partir da informação coletada durante o teste. Diante deste fato, ele se depara com duas questões a serem

respondidas:

- Quais dos requisitos de teste selecionados são reveladores de erros?
- Como selecionar novos requisitos de teste a serem investigados que forneçam mais indicações para a localização do defeito, ou o próprio sítio do defeito, a partir dos *rt-re*s identificados?

Note-se que as duas perguntas acima estão relacionadas com os dois passos do paradigma DDT *não* apoiados pelas técnicas de teste de depuração baseada em infomação de teste (Subseção 2.5.5). A primeira pergunta revela a necessidade de mecanismos que apóiem a *avaliação dos possíveis sintomas internos* (passo 3) representados pelos requisitos selecionados utilizando pares (heurística tipo *ranking*, requisito de teste). A segunda pergunta, por sua vez, indica que são necessários mecanismos de *seleção de novos possíveis sintomas internos* (passo 4), isto é, novos requisitos a serem investigados, a partir dos *rt-re*s identificados. A idéia é que o mantendeor pela identificação sucessiva de *rt-re*s termine por localizar o defeito.

No próximo capítulo, são introduzidos dois novos mecanismos que visam ajudar o mantenedor a responder as duas perguntas acima. Estes dois mecanismos utilizam informação coletada durante o teste e, por isso, têm um custo muito baixo em tempo de depuração. O uso de heurísticas baseadas em requisitos de teste juntamente com os mecanismos propostos estabelece uma estratégia de depuração que apóia *todos* os passos do paradigma DDT.

Capítulo 5

Uma Estratégia de Depuração Baseada em Requisitos de Teste

Neste capítulo, são introduzidos dois mecanismos cujo objetivo é permitir o uso de informação de teste em todos os passos do paradigma DDT. O primeiro mecanismo possibilita que o mantenedor monitore eventos relacionados às instâncias de um requisito candidato a revelar erros. Usando o monitoramento de eventos de teste, ele pode investigar se o requisito inclui o sítio do defeito no trecho de código associado ou se ele apenas fornece indicações úteis para a depuração. Dessa análise, o mantenedor pode concluir ainda que o requisito não contribui com informação relevante para a localização do defeito. O segundo mecanismo permite a seleção de novos requisitos de teste candidato a revelar erros a partir das indicações fornecidas pelos requisitos reveladores de erros. O uso dos pares (H,C) juntamente com estes dois mecanismos definem uma *estratégia de* **D***epuração baseada em* **R***equisitos de* **T***este* (**DRT**).

5.1 Motivação

O experimento descrito no capítulo anterior indicou que os pares (H,C) mais promissores são melhores na identificação de requisitos reveladores de erros (*rt-re*s) do que na seleção de um trecho de código que inclui o defeito. Porém, para a localização propriamente dita do sítio do defeito, duas questões precisam ser respondidas:

- Quais dos requisitos de teste selecionados são reveladores de erros?
- Como selecionar novos requisitos de teste que forneçam mais indicações para a localização do defeito, ou o próprio sítio do defeito, a partir dos *rt-re*s identificados?

Para auxiliar o mantenedor a responder a estas duas questões são propostos dois mecanismos de apoio ao uso de informação de teste na depuração. O primeiro mecanismo consiste no monitoramento

de eventos de teste associados às instâncias dos requisitos. Por meio do monitoramento de eventos de teste, o mantenedor pode investigar se o requisito é revelador de erro ou não. O resultado dessa investigação pode indicar que o sítio do defeito está localizado no trecho de código associado ao requisito ou que suas instâncias fornecem indicações úteis para a localização do defeito ou mesmo que o requisito não provê nenhuma informação relevante para a depuração. Esse mecanismo é implementado utilizando os resultados da análise de adequação e os recursos de depuradores simbólicos comuns.

O segundo mecanismo utiliza as indicações fornecidas por requisitos que revelam erros, mas não incluem o sítio de defeito, para selecionar novos requisitos candidatos a revelar erros. A idéia é que o mantenedor possa determinar o sítio do defeito identificando sucessivamente os requisitos reveladores de erros, presentes nos requisitos selecionados pelo mecanismo. É mostrado que isto ocorre quando o mantenedor é capaz de identificar corretamente os requisitos reveladores de erro.

O uso de heurísticas para determinação do conjunto inicial de requisitos candidatos a revelar erros juntamente com estes dois mecanismos definem uma *estratégia de* **D***epuração baseada em* **R***equisitos de* **T***este* (**DRT**). A característica mais interessante desta estratégia está no fato de utilizar informação de teste em todos os passos do paradigma DDT. Este fato faz com que ela tenha baixo custo de aplicação em tempo de depuração (veja discussão na Seção 6.4).

Na próxima seção, são descritos os eventos associados aos requisitos de teste rastreados pelo mecanismo de monitoramento. Na Seção 5.3, o mecanismo de seleção de requisitos de teste candidatos a revelar erros é apresentado. A Seção 5.4 descreve a estratégia **DRT**. A Seção 5.5 contém uma discussão sobre a estratégia introduzida; e a Seção 5.6 as considerações finais.

5.2 Definindo e Monitorando Eventos de Teste

Suponha-se que o mantenedor, com o objetivo de localizar defeitos em um programa P, utilize o par (H,C) para selecionar os requisitos de teste exercitados por um conjunto de casos de teste T candidatos a revelar erros. Os nós que fazem parte dos requisitos de teste selecionados definem um pedaço do código de P candidato a incluir o sítio do defeito. Duas situações podem ocorrer com relação à localização do defeito: ele pode não estar localizado no trecho identificado ou o mantenedor ao inspecionar o código pode não ser capaz de localizá-lo.

Essas situações implicam que, para obter-se informação útil para a depuração, os requisitos de teste selecionados deverão ser investigados para determinar aqueles que indicam possíveis sintomas internos, ou melhor, revelam erros. No entanto, para investigar estes requisitos de teste, o mantenedor deverá verificar o estado do programa quando determinados *eventos* associados às instâncias dos requisitos selecionados ocorrem. Por exemplo, ao se investigar um requisito de teste de fluxo de dados, é necessário verificar os valores das variáveis do requisito quando ocorrem dois eventos distintos: o alcance da definição e o alcance do (potencial) uso.

Entretanto, nem toda execução do comando onde ocorre a definição ou o (potencial) uso de uma

Tabela 5.1: Requisitos selecionados pelo par (H3, todos potenciais usos) aplicado à versão de número 11 no cenário MT-C.

merge()	
1	(3,(16,18),{ f })
3	(8,(16,18),{ ibuf })
4	(9,(15,16),{ j })
5	(10,(16,18),{ p })
7	(12,(16,18),{ i })
8	(13,(16,18),{ cp })
9	(14,(16,18),{ l })
10	(15,(11,12),{ k })
11	(15,(21,11),{ k })
12	(15,(13,20),{ k })
13	(15,(14,19),{ k })
14	(15,(14,15),{ k })
15	(15,(16,18),{ k })
16	(16,(16,18),{ j })
17	(18,(11,12),{ j })
18	(18,(21,11),{ j })
19	(18,(13,20),{ j })
20	(18,(14,19),{ j })
21	(18,(15,16),{ j })

variável está associada às instâncias do requisito. Por exemplo, considerem-se os requisitos apresentados na Tabela 5.1 que foram selecionados pela aplicação do par (H3, todos potenciais usos) no cenário MT-C para localizar o defeito da versão de número 11 do programa `Sort` (veja Tabela 4.2). A Figura 5.1 contém o trecho de código do programa `Sort` associado aos requisitos selecionados. O comando `++j;` é incluído neste trecho de código porque está associado ao nó 9 que, por sua vez, faz parte da *adpu* `(9,(15,16),j)`. Note-se que as execuções do comando `++j;` (vinculadas às instâncias do nó 9) que devem ser investigadas são aquelas cujo valor atribuído a `j` é potencialmente usado nas instâncias do ramo (15,16). As demais execuções não estão relacionadas com instâncias de `(9,(15,16),j)` e, por isso, não têm interesse para a depuração baseada em informação de teste. Isto ocorre porque o que está associado à ocorrência de falhas é o exercício de `(9,(15,16),j)` e não a simples execução do comando do nó 9. Daí a necessidade de verificar-se precisamente os eventos de teste.

A seguir são definidos os eventos de teste associados às instâncias dos requisitos de critérios de fluxo de controle e de dados. É apresentado também um exemplo de utilização da ferramenta `gdb/poke` que implementa o monitoramento de eventos de teste para auxiliar o mantenedor a identificar os requisitos de teste reveladores de erros.

```
merge(a,b)
/*  1 */        {
...
/*  3 */           {
/*  3 */              f = setfil(i);
/*  3 */              if(f == 0)
...
/*  8 */              ibuf[j] = p;
/*  8 */              if(! (fgets(( p )->l, 2048 , ( p )->b) == 0 ) )
/*  9 */                 j++;
/* 10 */              p++;
/* 10 */           }
/* 11 */           do
/* 11 */           {
/* 11 */              i = j;
/* 11 */              qsort((char **)ibuf, (char **)(ibuf+i));
/* 11 */              l = 0;
/* 12 */              while(i--)
/* 13 */              {
/* 13 */                 cp = ibuf[i]->l;
/* 13 */                 if(*cp == '\0')
/* 14 */                 {
/* 14 */                    l = 1;
/* 14 */                    if((fgets(( ibuf[i] )->l, 2048 , ( ibuf[i] )->b) ==
                                       ((void *)0) ) )
/* 15 */                    {
/* 15 */                       k = i;
/* 16 */                       while(++j <  k) /* <- sítio do defeito */
/* 17 */                          ibuf[k-1] = ibuf[k];
/* 18 */                       j--;
/* 18 */                    }
/* 19 */                 }
/* 20 */              }
/* 21 */           }
/* 21 */           while(l);
...
/* 62 */        }
```

Figura 5.1: Trecho de código associado aos requisitos de teste selecionados pela aplicação do par (H3, todos potenciais usos) no cenário MT-C para localizar o defeito da versão de número 11 do programa `Sort`.

5.2.1 Definição dos Eventos de Teste

Os eventos de teste ocorrem durante a execução de um caso de teste e estão relacionados com as instâncias dos requisitos de um critério C. Para acessar os eventos de teste e seus atributos são definidas *operações*. A seguir, os eventos, seus atributos e suas operações são definidos de forma geral.

Atributos

- **ponto de parada**: define o ponto no código onde um evento se completa e os atributos associados a ele podem ser observados;
- **condição**: define a condição que deve estar válida no *ponto de parada* para que o evento se complete;
- **valores dos dados**: valores das variáveis associadas ao requisito de teste.

Operações

- **first_<evento>**: pára na primeira ocorrência do evento de teste;
- **next_<evento>**: pára na próxima ocorrência do evento de teste;
- **last_<evento>**: pára na última ocorrência do evento de teste;
- **instance_<evento>**: pára em uma ocorrência particular do evento de teste;
- **next_check**: verifica a ocorrência de um evento de teste quando o código do programa é executado passo a passo.

Para descrever tanto os eventos de teste como o mecanismo de seleção de requisitos candidatos a revelar erros, de agora em diante, são retomadas algumas das suposições do Capítulo 4. Considera-se $\mathcal{S}$ um software composto de procedimentos e funções que contém um defeito revelado pelo teste com critérios de fluxo de controle e de dados. Seja P um procedimento ou função de $\mathcal{S}$ tal que seu grafo de fluxo de controle associado é $G(N, R, e, s)$. Para esse particular procedimento ou função sob investigação, o caso de teste revelador de defeito t possui a trajetória 1^1,..., i^p, ..., m^q, n^{q+1}, ..., j^r, k^{r+1},... para uma das invocações de P.

As instâncias dos requisitos dos critérios de teste estrutural podem ser descritas em termos de eventos de teste e seus atributos. Com relação ao critério todos ramos, há um evento associado a cada instância (m^q, n^{q+1}) de um ramo (m, n), isto é, o *alcance do ramo* (m, n) é o evento de teste.

Alcance do ramo (m, n)

Atributos

- **ponto de parada** : na *entrada* do nó n.

- **condição**: o nó anteriormente executado antes de n deve ser m.
- **valores dos dados**: valores das variáveis w que possuem um p-uso no ramo (m, n).

Operações: *first_branch*, *next_branch*, *last_branch*, *instance_branch* e *next_check*.

Com relação aos critérios de fluxo de dados, considere-se um requisito de fluxo de dados (i, j, D) ou $(i,(j, k),D)$[1] e suas instâncias (i^p, j^r, D, $\mathcal{V}$, $\mathcal{V}'$) ou (i^p, (j^r, k^{r+1}), D, $\mathcal{V}$, $\mathcal{V}'$). Dois eventos de teste são associados a cada instância. O alcance da *definição* e o alcance do (potencial) *uso* do requisito. Esses eventos de teste são descritos como se segue:

Alcance da definição

Atributos

- **ponto de parada**: na *saída* da instância i^p.
- **condição**: a próxima instância de j ou k é j^r ou k^{r+1} e a trajetória $i^p, \ldots, j^r$ ou $i^p, \ldots, j^r$, k^{r+1} é livre de definição c.r.a. variáveis $v \in D$.
- **valores dos dados** : conjunto de valores $\mathcal{V}$ das variáveis $v \in D$.

Operações: *first_def*, *instance_def*, *last_def* e *next_check*.

Alcance de uso

Atributos

- **ponto de parada**: na *entrada* da instância j^r ou k^{r+1}.
- **condição**: a última instância de i foi i^p e a trajetória $i^p, \ldots, j^r$ ou $i^p, \ldots, j^r$, k^{r+1} é livre de definição c.r.a. variáveis $v \in D$.
- **valores dos dados**: conjunto de valores $\mathcal{V}'$ das variavéis $v \in D$.

Operações: *first_use*, *next_use*, *instance_use*, *first_puse*, *next_puse*, *instance_puse* e *next_check*.

São ainda definidos dois outros eventos de teste relacionados com os requisitos de fluxo de dados: *alcance de uma redefinição* e *alcance de um ponto de influência*. Estes dois eventos são introduzidos para identificar a ocorrência de nós que influenciam os valores das variáveis do requisito de teste.

Alcance de uma redefinição

Atributos

[1] No caso de uma *adu* envolvendo um p-uso ou c-uso de uma variável v, $D = \{v\}$.

- **ponto de parada**: na *saída* de um nó h.
- **condição**: há uma definição de $v \in D$ em h e $h \neq i$.
- **valores dos dados**: valor das variáveis $v \in D$ que são redefinidas em h.

Operações: *next_check*, *next_use* e *next_puse*.

Alcance de um ponto de influência

Atributos

- **ponto de parada**: na *entrada* do nó n tal que (m, n) e $(m, n') \in R$.
- **condição**: n' domina um nó h, que não é dominado por n, tal que há em h uma definição de $v \in D$ e $h \neq i$.
- **valores dos dados**: valor das variáveis w que possuem um p-uso no ramo (m, n).

Operações: *next_check*, *next_use* e *next_puse*.

O alcance da redefinição altera o valor da variável associada ao requisito e evita que ele seja exercitado; por isso, este evento ocorre *entre* instâncias. O alcance de um ponto de influência, ao contrário, indica uma instância de um nó que, por ter sido alcançada, a variável do requisito *não* foi redefinida, permitindo o exercício do requisito. Esse evento busca identificar os pontos de *influência potencial* (Korel, 1988) (Subseção 2.5.4).

A seguir é apresentado um exemplo de utilização do monitoramento de eventos na localização de defeitos. O mecanismo proposto acima foi implementado na `gdb/poke` utilizando os resultados da análise de adequação dos critérios de teste apoiados pela ferramenta POKE-TOOL e os recursos do depurador simbólico `gdb` (Stallman e Pesch, 1999), conforme descrito no Capítulo 6.

5.2.2 Uso do Mecanismo de Monitoramento de Eventos de Teste

Considere-se novamente a versão de número 11 do programa `Sort` cujo trecho de código associado aos requisitos selecionados pelo par (H3, todos potenciais usos) no cenário MT-C é apresentado na Figura 5.1. O sítio do defeito está contido neste trecho e é impresso em negrito. Esse defeito possui as seguintes características: (1) é difícil de detectar pois, em 997 casos de teste da massa de teste, apenas oito revelam o defeito e (2) faz com que o programa entre em *loop*.

Suponha-se que o mantenedor ao inspecionar o código não tenha sido capaz de identificar o sítio do defeito — situação que pode ocorrer quando ele tem pouca familiaridade com o código (comum no cenário MT-C). O mantenedor pode então utilizar o mecanismo de monitoramento de eventos para identificar requisitos de teste reveladores de erros dentre os requisitos selecionados por meio de heurísticas.

Para exemplificar o uso do mecanismo de monitoramento de eventos de teste na gdb/poke, digamos que o mantenedor tenha decidido investigar se o requisito (18,(13,20),j) revela erros quando é exercitado por um caso de teste que provoca a ocorrência da falha. As convenções utilizadas a seguir são as seguintes: (gdb/poke) é o *prompt* da ferramenta; o texto escrito em **negrito** representa as entradas fornecidas pelo mantenedor; e o texto em fonte `máquina de escrever` indica os resultados fornecidos pela gdb/poke.

O comando breakassoc, descrito a seguir, prepara a gdb/poke para o monitoramento de eventos de teste de um requisito.

```
(gdb/poke) breakassoc -f merge -c pu < 18,(13,20), j>
Ready to breakpoint data-flow testing events.
```

Note-se que o comando acima ajusta a gdb/poke para monitorar eventos do requisito (18,(13,20), j) exigido pelo critério todos potenciais usos (indicado pela opção -c pu) na função merge() (indicada pela opção -f merge). Para verificar o valor de j e o estado do programa quando a primeira definição é alcançada, o mantenedor deve executar o comando first_def.

```
(gdb/poke) first_def
1613    /* 18 */                        j--;
First def instance has been reached at node 18 (18,1,186)
Variable j value: 7
```

Como resultado, a execução do programa pára na saída do nó 18 de forma que o valor de j que supostamente será usado é apresentado ao mantenedor. Para alcançar o p-uso da *adpu*, o comando next_puse deve ser executado:

```
(gdb/poke) next_puse
1590    /* 13 */                cp = ibuf[i]->l;
1591    /* 13 */                if(*cp == '\0')
P-use node has been reached at node 20 (20,1,196)
Variable j value: 7
Instance # 1
```

O comando next_puse pára na entrada do nó 20 e o valor de j que será usado é apresentado. O mantenedor pode verificar as próximas instâncias da *adpu* executando repetidamente do comando next_puse.

```
(gdb/poke) next_puse
1603    /* 16 */                        while(++j < k)
Variable j value: 8
A redefinition occurred at node 16 (16,1,229)
1613    /* 18 */                        j--;
```

```
Def node has been hit at node 18 (18,1,230)
Variable j value: 7
1590    /* 13 */                cp = ibuf[i]->l;
1591    /* 13 */                if(*cp == '\0')
P-use node has been reached at node 20 (20,1,235)
Variable j value: 7
Instance # 2
```

Note-se que a ferramenta evidencia a ocorrência de uma redefinição no nó 16 (evento alcance de uma redefinição). Para atingir a última instância do requisito sem passar pelas intermediárias, o mantenedor deve executar seguidamente os comandos `last_def` e `next_puse`.

```
(gdb/poke)  last_def
1613    /* 18 */                        j--;
Last def instance has been reached at node 18 (18,1,8164)
Variable j value: 7
(gdb/poke)  next_puse
1590    /* 13 */                cp = ibuf[i]->l;
1591    /* 13 */                if(*cp == '\0')
P-use node has been reached at node 20 (20,1,8169)
Variable j value: 7
Instance # 117
```

Observando as instâncias de `(18,(13,20),j)` nota-se que o valor de `j` definido e usado é o mesmo para todas as instâncias. Este fato indica que o valor de `j` está incorreto, pois a variável faz parte da condição de um laço e o programa foi interrompido por estar em laço infinito. Portanto, a *adpu* `(18,(13,20),j)` é reveladora de erro.

O mantenedor poderia executar passo a passo os caminhos das instâncias (controlando a ocorrência de eventos de teste) utilizando o comando `next_check` para determinar porque o valor de `j` é constante. No entanto, observando o alcance da redefinição no nó 16, evidenciado pelo comando `next_puse`, verifica-se que o valor `j` permanece constante porque é incrementado no nó 16. Logo, o sítio do defeito está localizado neste nó.

O mecanismo de monitoramento de eventos de teste da `gdb/poke` permite verificar facilmente as várias instâncias do requisito de teste candidato a revelar erros. Ele possibilita que o mantenedor investigue o estado do programa nas instâncias do nó de definição e de uso que estão associadas às instâncias do requisito, parando a execução nas *condições* e nos *pontos de parada* definidos para os eventos de teste. Essas características de monitoramento de eventos de teste requerem uma instrumentação especial do programa e o controle das instâncias dos requisitos de teste em tempo de execução. A implementação desse mecanismo é discutida na Subseção 6.3 do próximo capítulo.

5.3 Selecionando Requisitos de Teste Candidatos a Revelar Erros

O monitoramento de eventos de teste ajuda o mantenedor a identificar os requisitos reveladores de erros dentre aqueles selecionados, por exemplo, utilizando heurísticas. O defeito pode ser descoberto durante esse processo, pois ele pode estar localizado no código associado aos requisitos selecionados ou ele pode ser visitado quando as trajetórias das instâncias são examinadas.

Entretanto, como foi observado no Capítulo 4, muitas vezes o mantenedor vai conseguir apenas *indicações* durante essa análise. Portanto, para levar adiante o processo de localização, novos requisitos de teste candidatos a revelar erros devem ser selecionados a partir de requisitos reveladores de erro identificados. A identificação sucessiva de requisitos de teste reveladores de erro tem como objetivo guiar o mantenedor até o sítio do defeito. A seguir, é proposto um mecanismo para seleção de novos requisitos a serem investigados a partir das indicações fornecidas pelos requisitos reveladores de erro.

5.3.1 Situações Reveladoras de Erros

Inspecionando as definições do Capítulo 4, observa-se que as situações *reveladoras de erro* que fornecem apenas indicações (i.e., não incluem o sítio do defeito) podem ser classificadas em quatro tipos principais:

Valor Incorreto

Nessas situações, o requisito de teste é revelador de erro devido a um *valor incorreto* de uma variável x em uma instância i^p. Isto ocorre quando o valor de x é *usado* em uma expressão do nó i e o resultado da expressão é atribuído a uma das variáveis associadas ao requisito (Definição 5, situação 2) ou é utilizado na condição do comando de controle de fluxo condicional vinculado ao ramo do requisito (Definição 4, situação 1b). Na primeira situação, o requisito torna-se revelador de erro porque uma de suas variáveis recebe um valor incorreto (Definição 8, situação 1; Definição 9, situação 1) e, na segunda, porque o ramo associado é incorretamente executado (Definição 7, situação 1; Definição 9, situação 3).

Assim, os novos requisitos candidatos a revelar erros devem ser selecionados dentre as *adu*s e *adpu*s que envolvem o uso de x no nó i e que foram exercitadas pelo caso de teste revelador de defeito utilizado na sessão de depuração.

Valor de Retorno Incorreto

As situações desse tipo são semelhantes às descritas anteriormente. Os requisitos de teste são reveladores de erro porque uma função f(y) presente em uma expressão *retorna* um valor incorreto (Definição 4, situação 1c; Definição 5, situação 3). O resultado incorreto da expressão é então atribuído a uma variável do requisito ou usado para controlar a execução do ramo associado.

Nessas situações, os novos requisitos a serem investigados são as *adu*s e *adpu*s associadas às variáveis que são *usadas* na atribuição do valor de retorno de f(y) e exercitadas pelo caso de teste revelador de defeito utilizado na sessão de depuração.

Um requisito de teste pode ser revelador de erro porque uma instância j^r, que faz parte de uma instância do requisito, é incorretamente alcançada devido a uma das seguintes razões:

1. há uma instância (h^o, g^{o+1}), $o < r$, de um ramo (h, g) que engloba[2] j que foi executada quando não o deveria, levando à execução de j^r (Definição 3, situação 1); ou

2. o sítio do defeito é um comando de controle de fluxo incondicional incorreto em um nó h previamente alcançado em h^o, $o < r$, que provocou a execução de j^r (Definição 3, situação 2).

Com relação ao primeiro tipo de situação, o alcance incorreto da instância j^r é causado pela execução errônea de um ramo (h, g). Nesta situação, os próximos requisitos candidatos a revelar erros são selecionados dentre os ramos, as *adu*s e as *adpu*s exercitados pelo caso de teste revelador de defeito vinculados aos ramos que englobam imediatamente o nó j.

A segunda situação é tipicamente causada por comandos incondicionais incorretos. Por exemplo, no programa da Figura 4.1, o defeito de número 3 está localizado no comando `continue` que está no lugar de um comando `break`. Nessas situações, não é possível selecionar novos requisitos candidatos reveladores de erros visto que a localização do defeito exige a investigação da trajetória até j^r, em especial, dos comandos de controle de fluxo incondicionais executados.

Definição Ausente ou Efeito Colateral

Nessas situações, a instância do requisito de teste revelador de erro possui uma variável v cujo valor está *correto* na instância i^p do nó de definição; porém, na instância j^r do nó de (potencial) uso, o valor de v é *incorreto* (Definição 6, situação 1). Isto pode ocorrer quando:

1. o defeito é um comando ausente alcançado em m^q, $p < q < r$, que deveria conter uma definição de v; ou

2. há um procedimento (ou função) p() invocado na instância m^q que alterou o valor de v devido a um efeito colateral.

[2]Seja *suc(m)* o número de nós sucessores do nó m. Um ramo (m,n) *engloba* um nó k se *suc(m)* > 1 e n é *dominador* de k; este mesmo ramo engloba *imediatamente* k se não existe nenhum outro ramo que engloba k nos caminhos de n até k.

Na primeira situação (*Defeito Ausente*), a localização do defeito requer que a trajetória da instância seja investigada para a determinação do nó m visto que não é possível selecionar novos requisitos. O defeito de número 4 do programa da Figura 4.1 exemplifica essas situações.

No segundo tipo de situação (*Efeito Colateral*), o valor é alterado devido a um procedimento ou função invocado no caminho da instância. A variável v em questão é global, ou foi passada por referência, e teve o seu valor incorretamente modificado em p(). Por envolver fluxo de dados interprocedimental, nessas situações a depuração baseada em informação de teste requer a análise das instâncias dos requisitos de critérios de integração. Na Subsecão 5.5.2 discute-se como os mecanismos propostos são utilizados no contexto interprocedimental.

5.3.2 Seleção de Requisitos Candidatos a Revelar Erros

Nesta subseção são apresentados os conjuntos de requisitos de *teste de unidade* candidatos a revelar erros obtidos a partir das situações discutidas anteriormente.

Seleção refine_use

.

Com respeito às situações do tipo *Valor Incorreto*, o valor incorreto da variável x é usado na instância i^p. Os novos requisitos candidatos a revelar erros (*rt-c-re*$_{\text{Valor}}$) são então selecionados dentre as *adu*s e as *adpu*s exercitadas por um caso de teste $t \in T_R$ que envolvem o uso da variável x no nó i contidas no conjunto abaixo:

$$\textit{rt-c-re}_{\text{Valor}} = \{r \in RT_C(t) | t \in T_R \text{ e } r \text{ é da seguinte forma } (f, i, x) \text{ ou } (f, (h, g), D)\}$$

onde $C \in \{$todos usos, todos potenciais usos$\}$, $x \in D$, $f, h, g \in N$ e $(h, g) \in R$. Os requisitos de teste representados por $(f, (h, g), D)$ são selecionados dentre as *adu*s tal que $h = i$ e dentre as *adpu*s tal que (h,g) são ramos essenciais estendidos que alcançam imediatamente[3] i. Note-se que, para as *adpu*s, o mesmo valor de x é (potencialmente) usado no ramo (h,g) e no nó i (Maldonado, 1991; Maldonado et al., 1992b).

Um caso especial ocorre quando i^p é igual a 1^1 e x é um parâmetro formal. Suponha-se que o procedimento sob investigação tenha sido invocado pela última vez por um outro procedimento (ou função) p() em um nó i' do grafo de fluxo de controle de p(). Neste caso, os novos requisitos a serem investigados são selecionados da mesma maneira, porém, dentre as *adu*s e *adpu*s de p() vinculadas ao uso do parâmetro *real* amarrado a x no i'. Note-se que o procedimento p() e o nó i' são facilmente

[3]Um ramo (m,n) essencial estendido alcança imediatamente um nó k se não existe nenhum outro ramo desse tipo nos caminhos de n até k .

determinados em tempo de execução pois o estado da pilha de chamada (*call stack*) pode ser observado nos depuradores simbólicos.

O conjunto *rt-c-re*$_{\text{Valor}}$ pode ser obtido inspecionando o grafo de fluxo de controle e o conjunto $RT_C(t)$. Os algoritmos para determinação desse conjunto são apresentados no Capítulo 6. Como *rt-c-re*$_{\text{Valor}}$ contém a seleção dos usos candidatos a revelar erros, ele é chamado de seleção **refine_use**. A seguir, é apresentada uma argumentação (esboço de prova) de que **refine_use** seleciona pelo menos um requisito revelador de erro nas situações do tipo *Valor Incorreto*.

Observação 1 *Considere-se uma instância i^p nas situações do tipo Valor Incorreto. Se a seleção* **refine_use** *é diferente de vazia, então há entre os requisitos selecionados pelo menos um requisito (f,i,x) ou $(f,(h,g),D)$, onde $x \in D$, revelador de erro devido a uma instância (f^l,i^p,x,ν,ν') ou $(f^l,(h^o,g^{o+1}),D,\mathcal{V},\mathcal{V}')$, tal que $l \leq o < p$.*

Esboço de Prova:

De acordo com as situações do tipo *Valor Incorreto*, o valor de uma variável x usado em uma instância i^p é incorreto. A seleção **refine_use** seleciona as *adu*s e as *adpu*s exercitadas por um caso de teste $t \in T_R$ relacionadas com os usos de x em i. Se a seleção **refine_use** é não vazia, então pelo menos um dos requisitos selecionados (f,i,x) ou $(f,(h,g),D)$, onde $x \in D$, possui uma instância (f^l,i^p,x,ν,ν') ou $(f^l,(h^o,g^{o+1}),D,\mathcal{V},\mathcal{V}')$, tal que $l \leq o < p$, na qual o valor de x é incorreto em i^p ou g^{o+1} devido ao alcance prévio do sítio do defeito, ou seja, devido a um efeito do defeito (Definição 5, situação 2 e Definição 4, situação 1b). Logo, essa *adu* ou *adpu* é reveladora de erro (Definição 8, situação 2 e Definição 9, situação 2). •

Com relação às situações tipo *Valor de Retorno Incorreto*, o valor que a função f(y) retorna é incorreto; portanto, os novos requisitos de teste a serem investigados são as *adu*s e as *adpu*s da função f(y) exercitadas pelo caso de teste $t \in T_R$ que envolvem as variáveis *usadas* quando é atribuído o valor de retorno no nó de saída. Assim, o conjunto de novos requisitos a serem investigados é obtido da mesma forma que *rt-c-re*s_{Valor}; porém, neste caso, o nó i é o nó de saída de f(y) e a variável x representa todas as variáveis usadas na atribuição do valor de retorno de f(y).

Seleção refine_branch

.

As situações do tipo *Alcance Incorreto da Instância de um Nó* (referenciadas simplesmente por *Alcance Incorreto*) são dividas em dois subtipos. As situações de subtipo 1 são causadas pela execução incorreta de um ramo e possibilitam a investigação de novos requisitos. Já as situações de subtipo 2 são aquelas causadas por comandos de controle de fluxo incondicional errôneos. Para localizar estes defeitos é necessário investigar a trajetória até j^r, não sendo possível selecionar requisitos candidatos a revelar erros.

Assim, nas situações de subtipo 1, os requisitos candidatos a revelar erros são selecionados dentre os ramos, as *adu*s e as *adpu*s exercitados pelo caso de teste $t \in T_R$ vinculados aos ramos que englobam imediatamente o nó j. O conjunto de novos requisitos a serem investigados *rt-c-re*$_{\text{Alcance}}$ é descrito abaixo:

$$\textit{rt-c-re}_{\text{Alcance}} = \{r \in RT_C(t) |\ t \in T_R \text{ e } r \text{ possui a forma } (h,g) \text{ ou } (f,(h,g),D)\},$$

onde $C \in$ {todos ramos, todos usos, todos potenciais usos}, $f, h, g \in N$ e $(h,g) \in R$ é um ramo que engloba imediatamente o nó j.

Observe-se que pode ocorrer de o ramo (h,g) não ser essencial estendido, de forma que nenhuma *adpu* é selecionada. Por isso, para garantir que novos requisitos sempre serão selecionados, os requisitos dos critérios Potenciais Usos devem ser utilizados sempre em conjunto com os requisitos do critério todos ramos durante o processo de localização.

Analogamente às situações do tipo *Valor Incorreto*, j^r pode ser igual à instância 1^1 do procedimento sob investigação. Neste caso, os novos *rt-c-re*s são obtidos da mesma maneira descrita acima, porém a partir do nó i' do procedimento p().

Como o conjunto *rt-c-re*$_{\text{Alcance}}$ requer a determinação dos ramos que englobam imediatamente um nó, a seleção de requisitos estabelecida por este conjunto é chamada de **refine_branch**. Os algoritmos utilizados para determinar *rt-c-re*$_{\text{Alcance}}$ são igualmente apresentados no Capítulo 6. Abaixo é apresentada a argumentação de que **refine_branch** seleciona pelo menos um requisito revelador de erro nas situações do tipo *Alcance Incorreto* subtipo 1.

Observação 2 *Considere-se uma instância j^r nas situações do tipo Alcance Incorreto subtipo 1. Se a seleção* **refine_branch** *não é vazia, então há entre os requisitos selecionados um ramo (h,g) revelador de erro ou uma adu ou adpu $(f,(h,g),D)$ reveladora de erro devido a uma instância (h^o,g^{o+1}) ou $(f^l,(h^o,g^{o+1}),D,\mathcal{V},\mathcal{V}')$, tal que $l \leq o < r$.*

Esboço de Prova:

Nas situações do tipo *Alcance Incorreto* subtipo 1, a instância j^r é alcançada de forma incorreta, mas não devido à execução de um comando de controle de fluxo incondicional incorreto. Logo, a instância j^r somente é alcançada porque há um ramo (h,g) que engloba imediatamente j cuja instância (h^o,g^{o+1}), tal que $o < r$, causa a ocorrência j^r. Porém, se a instância (h^o,g^{o+1}) causou j^r, então ela também ocorreu em decorrência de um efeito do defeito, seja porque a expressão que controla (h,g) produziu um resultado incorreto (Definição 4, situação 1), seja porque h^o foi incorretamente alcançada (Definição 4, situação 2).

A seleção **refine_branch** seleciona os ramos que englobam imediatamente j e as *adu*s e as *adpu*s associadas a eles exercitados por um caso de teste $t \in T_R$. Como existe um ramo (h,g) que engloba imediatamente j cuja instância (h^o,g^{o+1}) ocorreu em decorrência de um efeito do defeito, então o ramo

(h,g), selecionado por **refine_branch**, é revelador de erro (Definição 7, situação 1). Da mesma maneira, as *adu*s ou *adpu*s (f,(h,g),D), selecionadas por **refine_branch**, são reveladoras de erro por causa da instância (f^l,(h^o,g^{o+1}),D,$\mathcal{V}$,$\mathcal{V}'$), tal que $l \leq o < p$ (Definição 9, situação 3). •

5.3.3 Resultados da Identificação Sucessiva de Requisitos Reveladores de Erro

O processo de identificação sucessiva de requisitos de teste de unidade reveladores de erro presentes nas seleções **refine_use** e **refine_branch** termina quando não é possível identificar novos requisitos a serem investigados. As situações reveladoras de erro que não permitem a seleção de novos requisitos são:

A) a instância (m^q,n^{q+1}) é executada porque o sítio do defeito está localizado na expressão expr(x,f(y), ...) do nó m (Definição 4, situação 1a);

B) a variável v possui um valor incorreto em i^p porque o sítio do defeito está localizado na expressão v = expr'(x,f(y), ...)) do nó i (Definição 5, situação 1);

C) o sítio do defeito é um comando de controle de fluxo incondicional incorreto que levou à execução de j^r (Situação tipo *Alcance Incorreto da Instância de um Nó*, subtipo 2);

D) a trajetória i^p,...,m^q,n^{q+1},...,j^r,k^{r+1} da instância do requisito (i, j, v) ou $(i,(j, k),D)$ é tal que o sítio do defeito é um comando ausente no nó m de forma que o valor em j^r é ν' quando deveria ser ν (Situação *Definição Ausente*);

E) a variável x possui um valor incorreto em i^p devido ao alcance prévio do sítio do defeito e a definição do valor usado em i^p ocorreu em outro procedimento (Situações *Valor Incorreto* e *Efeito Colateral*).

Note-se que a identificação de requisitos reveladores de erro devido às situações do tipo A e B implica que o defeito foi localizado. Já nas situações do tipo C e D, a informação de teste que leva à localização do defeito é a trajetória; portanto, o mantenedor deve examiná-la para determinar o sítio do defeito propriamente dito. As situações E, por sua vez, envolvem aqueles requisitos reveladores de erro cujas indicações estão relacionadas com fluxos de dados interprocedimentais; logo, os requisitos de teste de unidade não podem fornecer indicações adicionais para depuração. Nessas situações, é necessário utilizar os requisitos de teste de integração (veja discussão na Subseção 5.5.2) para prosseguir a depuração baseada em informação de teste.

A seguir, é apresentada uma argumentação (esboço de prova) de que, a partir de um conjunto inicial de requisitos candidatos a revelar erros (*rt-c-re*s) e utilizando sucessivamente as seleções **refine_use** e **refine_branch**, o mantenedor *pode* ser guiado até a identificação de requisitos reveladores de erro devido a uma das razões descritas nas situações tipo A, B, C, D e E. Para a argumentação que se segue, são assumidas *condições* necessárias para a localização do defeito a partir de requisitos de teste e a existência de um *oráculo confiável* de depuração.

Definição 15 Para a *localização de defeitos* utilizando requisitos de teste é necessário que as seguintes *condições* sejam válidas:

1. durante o teste de unidade com os critérios todos ramos, todos usos e todos potenciais usos foi obtido um conjunto de casos de teste reveladores de defeito T_R e coletados os resultados de teste;

2. o conjunto inicial de requisitos candidatos a revelar erros inclui pelo menos um que é revelador de erro.

Definição 16 Define-se como *oráculo confiável de depuração* o mantenedor capaz de identificar as instâncias dos requisitos de teste que revelam erros utilizando o monitoramento de eventos discutido na seção anterior.

Dessa maneira, seja *sel* = {**refine_use**,**refine_branch**} tal que *sel*(r) representa o conjunto de requisitos selecionados utilizando **refine_use** ou **refine_branch** a partir do requisito revelador de erro $r \in E_{(C,t)}$, onde $E_{(C,t)}$ é o conjunto de requisitos reveladores de erro exercitados por um caso de teste $t \in T_R$ e $C \in$ {todos ramos, todos usos, todos potenciais usos}.

Teorema 1 *Para os tipos de defeito considerados na Definição 1 e nas condições da Definição 15, a aplicação sucessiva de* **refine_use** *e* **refine_branch** *por um oráculo confiável de depuração leva à identificação de um requisito revelador de erro devido a uma das situações descritas nos itens A, B, C, D e E.*

Esboço de Prova:

Considere-se a aplicação sucessiva de **refine_use** e **refine_branch** *sel*$_0(r_0)$, ..., *sel*$_i(r_i)$, ..., *sel*$_k(r_k)$ tal que r_0, ..., r_i, ..., $r_k \in E_{(C,t)}$ e $k \geq 0$. Para mostrar que k é limitado, suponha-se, ao contrário, que seu valor seja infinito, isto é, o mantenedor (oráculo confiável) seleciona requisitos candidatos a revelar erros indefinidamente. Entretanto, segundo a Observação 1, a seleção **refine_use** aplicada a partir da instância i^p inclui pelo menos um requisito revelador de erro em que a instância f^l do nó de definição ocorre *antes* de i^p na trajetória do caso de teste t, pois $l < p$. Da mesma maneira, na Observação 2, verifica-se que a instância (h^o,g^{o+1}) ocorre *antes* de j^r na trajetória visto que $o < r$. Portanto, a cada seleção *sel*$_i(r_i)$, o mantenedor confiável identifica instâncias que revelam erros em posições anteriores na trajetória de $t \in T_R$. Assim, se k é infinito, isto implica que seriam identificadas instâncias reveladoras de erro cujas ocorrências dos nós são anteriores à instância 1^1, o que é uma contradição.

Analogamente, suponha-se que nenhum dos requisitos reveladores de erro identificados depois da aplicação sucessiva de **refine_use** e **refine_branch** o sejam devido a uma das situações do tipo A, ..., E. Como as situações do tipo A, B, C e D não ocorrem em nenhum dos requisitos reveladores de erro identificados depois da aplicação de *sel*$_k(r_k)$, então o sítio do defeito não está localizado no código

Tabela 5.2: Requisitos selecionados pelo par (H3, todos usos) aplicado à versão de número 4 no cenário GT-C.

	cmp()
1	(3,(64,66), pa)
2	(3,(68,73), pa)
3	(3,75, pa)
4	(3,(66,68), pb)
5	(3,(73,75), pb)
6	(3,75, pb)
7	(65,(68,69), pa)
8	(67,(66,67), pb)
9	(67,67, pb)
10	(67,(69,71), pb)
11	(75,(64,65), pa)
12	(75,65, pa)
13	(75,(66,67), pb)
14	(75,67, pb)

dos procedimentos analisados. Portanto, o defeito está localizado em um outro procedimento, não investigado. Porém, também não há requisitos reveladores de erro devido às situações do tipo E; logo, o defeito localizado no procedimento *não investigado não* influencia os requisitos dos procedimentos *investigados*. Isto é uma contradição visto que o mantenedor é um oráculo confiável de depuração e o caso de teste utilizado é revelador de defeito. •

As Observações 1 e 2 mostram que as seleções **refine_use** e **refine_branch** indicam novos possíveis sintomas internos — requisitos reveladores de erros — a partir de sintomas previamente identificados; portanto, apóiam o passo 4 do paradigma DDT. O Teorema 1, por sua vez, mostra que a utilização recorrente de **refine_use** e **refine_branch** *pode* guiar o mantenedor até a localização do defeito, desde que ele consiga identificar os requisitos reveladores de erro presentes nos conjuntos de novos requisitos a serem investigados. O monitoramento de eventos auxilia o mantenedor nessa tarefa.

5.3.4 Uso do Mecanismo de Seleção de Requisitos de Teste

A seguir é apresentado um exemplo de utilização do mecanismo de seleção de requisitos candidatos a revelar erros implementados na `gdb/poke`. O mecanismo será utilizado, juntamente com o mecanismo de monitoramento de eventos, para auxiliar o mantenedor a localizar o defeito inserido na versão de número 4 do programa `Sort` (veja Tabela 4.2). O conjunto inicial de requisitos de teste candidatos a revelar erros é obtido pela aplicação do par (H3, todos usos) no cenário GT-C e é apresentado na Tabela 5.2. A Figura 5.2 contém o trecho de código associado aos requisitos selecionados, bem como o sítio do defeito (identificado pelo fonte em **negrito**) que *não* faz parte do código selecionado pela heurística.

Suponha-se que o mantenedor decida verificar se a *adu* (65,(68,69), pa) é reveladora de erro utilizando a gdb/poke.

```
(gdb/poke) breakassoc -f cmp -c uses < 65,(68,69), pa>
Ready to breakpoint data-flow testing events.
```

Utilizando os comandos first_def e next_puse o mantenedor pode verificar o estado do programa na primeira instância.

```
(gdb/poke) first_def
2382    /* 65 */                pa++;
First def instance has been reached at node 65 (5,65,25)
Variable pa value: 0x805cdf6 ""

(gdb/poke) next_puse
2398    /* 68 */             if(pa>=la || *pa=='\n')
2402    /* 69 */                if(pb<lb && *pb!='\n')
P-use node has been reached at node 69 (5,69,29)
Variable pa value: 0x805cdf6 ""
Instance # 1
```

Se o mantenedor quiser avançar mais rapidamente em direção às outras instâncias, por exemplo, a décima terceira instância, ele pode executar o comando instance_puse.

```
(gdb/poke) instance_puse 13
2382    /* 65 */                pa++;
Def node has been hit at node 65 (188,65,23)
Variable pa value: 0x805d33e ""
2398    /* 68 */             if(pa>=la || *pa=='\n')
2402    /* 69 */                if(pb<lb && *pb!='\n')
P-use node has been reached at node 69 (188,69,27)
Variable pa value: 0x805d33e ""
Instance # 13
Use instance 13 has been reached
```

Utilizando last_def e next_puse é atingida a última instância do requisito.

```
(gdb/poke) last_def
Last def instance has been reached at node 65 (1934,65,37)
Variable pa value: 0x805cbc7 ""
(gdb/poke) next_puse
2398    /* 68 */             if(pa>=la || *pa=='\n')
2402    /* 69 */                if(pb<lb && *pb!='\n')
P-use node has been reached at node 69 (1934,69,41)
Variable pa value: 0x805cbc7 ""
Instance # 122
```

```
field()
{
...
      case 'd':
       p-> ignore = dict+290; /* <- sítio do defeito */
      break;
...
}

cmp(a,b)
/*  1 */        {
...
/*  3 */              fp = &fields[k];
/*  3 */              pa = i;
/*  3 */              pb = j;
/*  3 */              if(k)
/*  4 */              {
...
/* 64 */              while(ignore[*pa])
/* 65 */                 pa++;
/* 66 */              while(ignore[*pb])
/* 67 */                 pb++;
/* 68 */              if(pa>=la || *pa=='\n')
/* 69 */              {
/* 69 */                 if(pb<lb && *pb!='\n')
/* 70 */                 {
...
/* 70 */                 }
/* 71 */                 else
/* 71 */                    continue;
/* 72 */              }
/* 73 */              if(pb>=lb || *pb=='\n')
/* 74 */                 return(-fp->rflg);
/* 75 */              if((sa = code[*pb++]-code[*pa++]) == 0)
/* 76 */              {
...
/* 82 */        }
```

Figura 5.2: Trecho de código associado aos requisitos de teste selecionados pela aplicação do par (H3, todos usos) no cenário GT-C para localizar o defeito da versão de número 4 do programa `Sort` (o sítio do defeito é indicado mas não faz parte do trecho de código selecionado).

Os valores de pa são iguais tanto no alcance da definição como no alcance do uso para todas as instâncias verificadas. Aparentemente, as situações reveladoras de erro do tipo *Valor Incorreto* não ocorrem. Para verificar se as situações do tipo *Alcance Incorreto* (subtipo 1) estão presentes nas instâncias da *adu* (65,(68,69), pa), o mantenedor precisa investigar o alcance da definição nas instâncias. Usando a seleção **refine_branch**, ele pode selecionar requisitos que afetam o alcance do nó de definição de (65,(68,69), pa).

```
(gdb/poke)  refine_branch  -f cmp -c uses 65
 49 <3,(64,65), pa>
531 <62, (64,65), ignore>
535 <5,(64,65), pa>
549 <75,(64,65), pa>
```

O comando acima seleciona, dentre as *adu*s exercitadas pelo caso de teste revelador de defeito, aquelas cujos p-usos envolvem imediatamente o nó 65 — o nó de definição. Como o valor de pa está aparentemente correto, suponha-se que o mantenedor decida investigar a *adu* (62, (64,65), ignore).

```
(gdb/poke) breakassoc -f cmp -c uses < 62, (64,65), ignore>
Ready to breakpoint data-flow testing events.
(gdb/poke) rerun
Entered function cmp -- 5th invocation
(gdb/poke) first_def
2371    /* 62 */              code = fp->code;
2372    /* 62 */              ignore = fp->ignore;
First def instance has been reached at node 62 (5,62,6)
Variable ignore value: 0x804d862 ""
(gdb/poke) next_puse
2378    /* 64 */              while(ignore[*pa])
2382    /* 65 */                 pa++;
P-use node has been reached at node 65 (5,65,25)
Variable ignore value: 0x804d862 ""
Instance # 1
```

Logo na primeira instância, a variável ignore assume um valor incorreto, tanto na definição como no uso. Normalmente, o conteúdo dessa variável compreende os valores 1 e 0, sendo que o primeiro elemento do vetor é sempre 1. Portanto, está caracterizada a situação reveladora de erro do tipo *Valor Incorreto*, pois o valor de ignore está errado no nó 62. Para refinar ainda mais a busca, o mantenedor pode investigar os usos que ocorrem no nó 62 utilizando a seleção **refine_use**.

```
(gdb/poke) refine_use  -f cmp -c uses 62
85 <3,62, fp>
```

O comando **refine_use** acima seleciona as *adu*s cujos c-usos ocorrem em 62. Há apenas uma única *adu* exercitada e, portanto, a ser investigada.

```
(gdb/poke) breakassoc -f cmp -c uses < 3,62, fp>
Ready to breakpoint data-flow testing events.
(gdb/poke) rerun
Entered function cmp -- 1st invocation
(gdb/poke) first_def
2089    /*  3 */              fp = &fields[k];
2090    /*  3 */              pa = i;
2091    /*  3 */              pb = j;
2092    /*  3 */              if(k)
First def instance has been reached at node 3 (1,3,3)
Variable fp value: (struct field *) 0x804de60
(gdb/poke) print fp-> ignore
$4 = 0x804d862 ""
(gdb/poke) print fields[k].ignore
$5 = 0x804d862 ""
```

A variável `fp` aponta para a variável global `fields` que possui um valor incorreto no seu campo `ignore` na função `cmp()`. Observe-se que a *adu* `(3,62, fp)` é reveladora de erro devido ao uso do valor de uma variável global definida em um outro procedimento (situação tipo E). Como a `gdb/poke` utiliza os resultados da POKE-TOOL e esta ferramenta apóia apenas o teste de unidade, a seleção de novos requisitos a serem investigados não pode seguir adiante. Se os resultados do teste de integração estivessem disponíveis, o processo de localização poderia continuar analisando as *adu*s interprocedimentais da variável global `fields`.

Entretanto, inspecionando o código, pode-se observar que o campo `ignore` da variável `field` recebe valores em três pontos diferentes: nas linhas 1124, 2743 e 2757. Colocando *breakpoints* nessas linhas e o reexecutando o programa, obtém-se:

```
(gdb/poke) rerun
Breakpoint 2, field (s=0xbffffd4c "dfr", k=0) at testeprog.c:2743
2743    /*  6 */                    p->ignore = dict+290;
(gdb/poke)  print p-> ignore
$1 = 0x804e0a0 ""
```

O valor de `p->ignore` (o ponteiro `p` aponta para `fields`) é incorreto no nó 6 da função `field()` porque o deslocamento deveria ser 128, e não 290. Portanto, o defeito foi localizado.

5.4 DRT — Uma Estratégia de Depuração Baseada em Requisitos de Teste

5.4.1 Descrição da Estratégia

Nos exemplos apresentados nas subseções anteriores foi utilizada uma estratégia de depuração que combina as heurísticas discutidas no Capítulo 4 e os mecanismos propostos para a localização de defeitos. A seguir, esta estratégia é formalizada:

1. Utilize heurísticas para identificar um conjunto inicial de requisitos candidatos a revelar erros (*rt-c-re*s).

2. Verifique se o trecho de código associado aos *rt-c-re*s contém o sítio do defeito.

3. **Se** o sítio do defeito está localizado no código associado aos *rt-c-re*s **então** o defeito foi encontrado e o processo de localização está terminado.

4. **Caso contrário**, verifique se os *rt-c-re*s revelam erros utilizando o mecanismo de monitoramento de eventos de teste.

5. **Se** um requisito revelador de erro (*rt-re*) foi identificado **então**

   5.a) **Se** o sítio do defeito está localizado no código do *rt-re* ou foi localizado durante a análise das trajetórias das instâncias **então** o defeito foi encontrado e o processo de localização está terminado.

   5.b) **Caso contrário**, a partir das indicações obtidas e utilizando *refine_use* ou *refine_branch*, selecione um novo conjunto de requisitos de teste candidatos a revelar erros (*rt-c-re*s) e retorne ao passo 2.

6. **Caso contrário**, o processo de localização baseada no uso de informação de teste falhou.

Note-se que o uso da estratégia **DRT** não garante que o defeito seja localizado. Primeiro porque ela se baseia em informação heurística — o conjunto inicial de *rt-c-re*s pode não conter requisitos reveladores de erro. Segundo porque o mantenedor *real* não é um oráculo de depuração confiável, podendo acontecer dele supor incorretamente que um requisito é revelador de erro e direcionar o processo de localização para a análise de requisitos de teste que não levarão ao sítio do defeito.

5.5 Discussão

5.5.1 Depuração Guiada por Informação de teste

O monitoramento de eventos é um conceito bem comum no contexto de depuração, em especial, na depuração baseada em rastreamento e inspeção (Subseção 2.5.1). Entretanto, o monitoramento proposto é diferente, pois utiliza os resultados do teste estrutural para rastrear tanto os eventos como as trajetórias relacionados às instâncias dos requisitos de teste.

A idéia é auxiliar o mantenedor a investigar as instâncias de um requisito de teste para determinar se ele é revelador de erro ou não. Note-se que esta investigação é uma tarefa muito difícil, ou mesmo impossível, utilizando os depuradores simbólicos comuns. Por exemplo, na sessão de depuração apresentada no segundo exemplo, a *adu* `(65, (68,69),pa)` possui 122 instâncias, sendo que a última instância ocorre na invocação de número 1934 da função `cmp()`. Com as funcionalidades introduzidas na `gdb/poke`, esta e outras instâncias podem ser facilmente investigadas.

Além disso, o monitoramento de eventos de teste pode ser útil mesmo quando o trecho de código inclui o sítio do defeito. Como observado no primeiro exemplo, o mantenedor pode não ser capaz de determinar o sítio do defeito pela simples inspeção do trecho de código selecionado. Isto ocorre porque muitas vezes é necessário verificar os valores dos atributos de dados e o estado do programa nos eventos de teste para localizar o defeito. Segundo Korel e Rilling (1997), a representação textual de uma *fatia* (e.g., o código associado aos requisitos selecionados) não fornece informação para o entendimento do comportamento do programa, sendo freqüentemente este o fator mais importante para a depuração eficiente do programa. Neste sentido, o monitoramento de eventos de teste permite o exercício de um requisito, ajudando, dessa maneira, o mantenedor a *entender* o trecho de código selecionado e a localizar o defeito.

A necessidade de *entendimento* da dinâmica dos requisitos de teste é mais evidente quando o trecho de código selecionado não inclui o sítio do defeito. Nesses casos, o mantenedor deve *exercitar* os requisitos de teste para identificar as situações adequadas para a utilização do mecanismo de seleção de requisitos candidatos a revelar erros (seleções **refine_use** e **refine_branch**). As seleções **refine_use** e **refine_branch** identificam requisitos de teste que *podem* afetar um requisito revelador de erro e que foram exercitados pelo caso de teste da sessão de depuração. Neste sentido, a informação fornecida não é precisa como a de uma *fatia* dinâmica. A vantagem desta abordagem, entretanto, é que seu custo é baixo em tempo de depuração visto que a informação utilizada foi obtida no teste (veja Subseção 6.4).

É importante observar que os dois mecanismos propostos — monitoramento de eventos de teste e seleção de novos requisitos a serem investigados — atuam em conjunto no processo de localização do defeito. No último exemplo, analisando as instâncias da *adu* `(65, (68,69),pa)`, há uma suspeita de que a definição é incorretamente alcançada. A seleção **refine_branch** é então utilizada para confirmar esta hipótese, selecionando novos requisitos a serem investigados, o que requer novamente o uso do

monitoramento de eventos de teste para análise das novas hipóteses levantadas.

As funcionalidades fornecidas pela `gdb/poke` são similares às das ferramentas PELAS (Korel, 1988; Korel e Rilling, 1997, 1998), Spyder (Agrawal, 1991; Viravan, 1994) e FIND (Shimomura, 1993; Shimomura et al., 1995). Essas ferramentas permitem que o mantenedor verifique pontos de execução anteriores que afetaram o valor de uma variável. Dessa forma, elas possibilitam o refinamento da busca pelo defeito pois indicam outros possíveis pontos onde ele pode estar localizado.

A estratégia **DRT** difere das anteriores porque se baseia em informação coletada durante o teste, daí a razão do seu baixo custo. A ferramentas PELAS, Spyder e FIND utilizam informação coletada em tempo de depuração. Como conseqüência, elas não podem acessar instâncias dos requisitos de teste e possuem limitações em termos de escalabilidade. Como discutido no Capítulo 2, essas ferramentas utilizam grafos cuja construção é custosa (em termos de espaço e tempo) para programas reais. No próximo capítulo o custo dos algoritmos da `gdb/poke` é comparado com o custo de ferramentas similares.

5.5.2 Requisitos de Teste de Integração

Como evidenciado na sessão de depuração apresentada no último exemplo, nas situações do tipo E, a localização do defeito utilizando a estratégia **DRT** somente pode prosseguir se os requisitos de teste de integração estiverem disponíveis. A seguir, é discutido o uso desses requisitos na estratégia.

Os critérios de teste de fluxo de dados, originalmente definidos para o teste de unidade, foram estendidos para o teste de integração. Vários autores (Harrold e Soffa, 1991; Jin e Offutt, 1998; Linnenkugel e Müllerburg, 1990) propuseram critérios de fluxo de dados interprocedimentais semelhantes aos da família Fluxo de Dados; Vilela (1998), por sua vez, utiliza o conceito potencial uso no teste de integração.

No contexto de integração, os conceitos de *instância* e *requisito revelador de erro* podem ser utilizados com poucas modificações. Sejam P e Q procedimentos do software $\mathcal{S}$ cujos grafos de fluxo de controle são $G_P(N_P, R_P, e_P, s_P)$ e $G_Q(N_Q, R_Q, e_Q, s_Q)$, respectivamente, tal que $i_P \in N_P$ e (j_Q, k_Q) $\in R_Q$. De maneira semelhante aos requisitos dos critérios de teste de unidade, um requisito de teste de integração (i_P, j_Q, D) ou (i_P, (j_Q,k_Q), D), em que a definição ocorre em P e o (potencial) uso em Q, possui instâncias definidas pelas quíntuplas (i_P^q, j_Q^r,D, $\mathcal{V}$,$\mathcal{V}'$) ou (i_P^q, (j_Q^r,k_Q^{r+1}),D, $\mathcal{V}$,$\mathcal{V}'$). Note-se que, para os critérios que estendem os critérios da família Fluxo de Dados, $D = \{v_P$,$v_Q\}$ onde v_Q é a variável em Q que faz referência a v_P. Analogamente, para o critérios Potenciais Usos de Integração, D $= \{v'_P$,v'_Q,v''_P,v''_Q,...$\}$ onde v'_P, v''_P são as variáveis definidas em P que são potencialmente usadas em D utilizando as referências a v'_Q, v''_Q. Se v_P é uma variável global, então v_P=v_Q=v. Assim, as situações que revelam erros para os requisitos de teste de integração são as mesmas definidas para os requisitos de teste de unidade (Definições 8 e 9).

Do ponto de vista de monitoramento de eventos, os requisitos de fluxo de dados de integração estão associados aos mesmos eventos de teste (com os respectivos atributos e operações) descritos na

Seção 5.2. A diferença está na implementação do monitoramento desses eventos que requer o uso de informação de fluxo de dados global. Na Subseção 6.3 discute-se como monitorar esses eventos.

Com relação ao mecanismo de seleção de requisitos, apenas a seleção **refine_use** é alterada quando são consideradas as situações do tipo E. Note-se que essas situações são equivalentes às do tipo *Valor Incorreto* pois o valor da variável x usado em i^p é incorreto. A diferença é que a definição de x ocorreu em outro procedimento, de forma que o processo de localização somente pode seguir adiante utilizando os requisitos de integração.

Assim, considerando-se i igual a i_P no contexto de integração e Q' um procedimento que chama P, o conjunto *rt-c-re*$_{INT}$ de requisitos de integração selecionados é definido da seguinte maneira:

$$\textit{rt-c-re}_{INT} = \{r \in RT_C(t) | t \in T_R \text{ e } r \text{ é da seguinte forma } (f_Q, i_P, D) \text{ ou } (f_Q, (h_P, g_P), D)\}$$

onde $C \in$ {todos usos de integração, todos potenciais usos de integração}, $x \in D$, $f_Q \in N_Q$, $i_P \in N_P$ e $(h_P, g_P) \in R_P$. Os requisitos de teste representados por $(f_Q, (h_P, g_P), D)$ são selecionados dentre as *adu*s tal que $h_P = i_P$ e entre as *adpu*s tal que (h_P,g_P) são ramos essenciais estendidos que alcançam imediatamente i_P.

Portanto, os requisitos de integração podem ser diretamente introduzidos na estratégia **DRT**, desde que disponíveis.

5.6 Considerações Finais

Neste capítulo, foram propostos dois mecanismos — monitoramento de eventos de teste e seleção de requisitos candidatos a revelar erros — cujo objetivo é apoiar o uso de informação de teste em todos os passos do paradigma DDT. O uso combinado das heurísticas para seleção inicial de requisitos candidatos a revelar erros com os mecanismos propostos define uma *estratégia de* **D***epuração baseada em* **R***equisitos de* **T***este* (**DRT**). Essa estratégia — apoiada por ferramentas que implementam as heurísticas e os mecanismos desenvolvidos (`pokedebugheur` e `gdb/poke`) — foi utilizada para localizar um dos defeitos introduzidos no programa `Sort` no Capítulo 4.

O objetivo do primeiro mecanismo é possibilitar que o mantenedor verifique os eventos e as trajetórias relacionados com as instâncias de um requisito de teste. Os eventos de teste foram propostos para identificar pontos da execução associados às instâncias dos requisitos de teste. Monitorando os eventos de teste, o mantenedor pode verificar dinamicamente se o defeito está localizado no código associado ao requisito ou se apenas indicações úteis para a depuração são fornecidas por suas instâncias. Dessa análise, o mantenedor pode concluir ainda que o requisito não contribui com nenhuma informação relevante para a localização do defeito. O mecanismo foi implementado na ferramenta `gdb/poke` utilizando os resultados da análise de adequação e os recursos comuns do depurador simbólico `gdb`.

O segundo mecanismo apóia o refinamento do conjunto de requisitos de teste candidatos a revelar erros por meio das seleções **refine_use** e **refine_branch**. Ele indica requisitos que foram exercitados pelo caso de teste e que podem afetar o requisito revelador de defeito. As seleções **refine_use** e **refine_branch**, quando aplicadas nas situações adequadas, movem o processo de localização adiante visto que incluem pelo menos um requisito de teste revelador de erro. A identificação sucessiva dos requisitos reveladores de erro presentes nos conjuntos selecionados por **refine_use** e **refine_branch** *pode* indicar um requisito tal que o sítio do defeito ou está no seu trecho de código ou associado à trajetória da instância que revela o erro.

Assim, pode-se observar que os resultados do teste guiam o processo de localização do defeito na estratégia **DRT**, de maneira que *todos* os passos do paradigma DDT são realizados tendo como suporte a informação de teste. A *identificação de possíveis sintomas internos* (passo 2) é feita pela seleção do conjunto inicial de requisitos candidatos a revelar erros por meio de heurísticas. A *avaliação dos possíveis sintomas internos* (passo 3) é realizada monitorando eventos de teste, o que somente é possível utilizando os resultados da análise de adequação. As seleções **refine_use** e **refine_branch**, por sua vez, identificam *novos possíveis sintomas internos* (passo 4) a partir dos requisitos exercitados por um caso de teste. Por basear-se em informação coletada durante o teste, a estratégia possui um custo muito baixo em tempo de depuração, tornando-a mais vantajosa do que estratégias similares. No próximo capítulo, a implementação da ferramenta `gdb/poke` é descrita em detalhes.

Capítulo 6

Aspectos de Implementação da gdb/poke

Neste capítulo são apresentados os aspectos mais relevantes da implementação do monitoramento de eventos de teste e da seleção de requisitos candidatos a revelar erros na ferramenta gdb/poke.

6.1 Implementação da gdb/poke

A gdb/poke é uma ferramenta desenvolvida a partir do depurador simbólico gdb (Stallman e Pesch, 1999) (no Apêndice C é apresentada uma breve discussão das principais funções do gdb) para apoiar a aplicação da estratégia **DRT** em programas escritos em C. Ela apóia a estratégia implementando os mecanismos para monitorar eventos e selecionar requisitos candidatos a revelar erros (*rt-c-re*). Para tanto, ela utiliza os resultados do teste estrutural e da aplicação de heurísticas baseadas em requisitos de teste, fornecidos, respectivamente, pela ferramenta POKE-TOOL e o módulo pokedebugheur. O relacionamento da gdb/poke com a POKE-TOOL e o módulo pokedebugheur é apresentado na Figura 6.1.

A monitoração de eventos de teste é realizada colocando-se *breakpoints* em pontos especiais de uma versão modificada do programa e usando *scripts* (escritos na linguagem fornecida pelo gdb) para localizar os eventos propriamente ditos. A seleção de requisitos candidatos a revelar erros, por sua vez, é feita consultando-se tanto os resultados da análise de adequação dos casos de teste como os resultados da análise estática do programa, realizada para determinar os requisitos de teste. A seguir, os principais aspectos de implementação da monitoração de eventos de teste e da seleção de requisitos são discutidos.

6.2 Instrumentação

A ferramenta gdb/poke utiliza uma versão especial do programa em teste para depuração. Esta versão contém o código necessário para verificar eventos de teste e é obtida invocando-se a POKE-TOOL com uma opção especial. O programa executável a ser depurado deve ser gerado a partir da versão

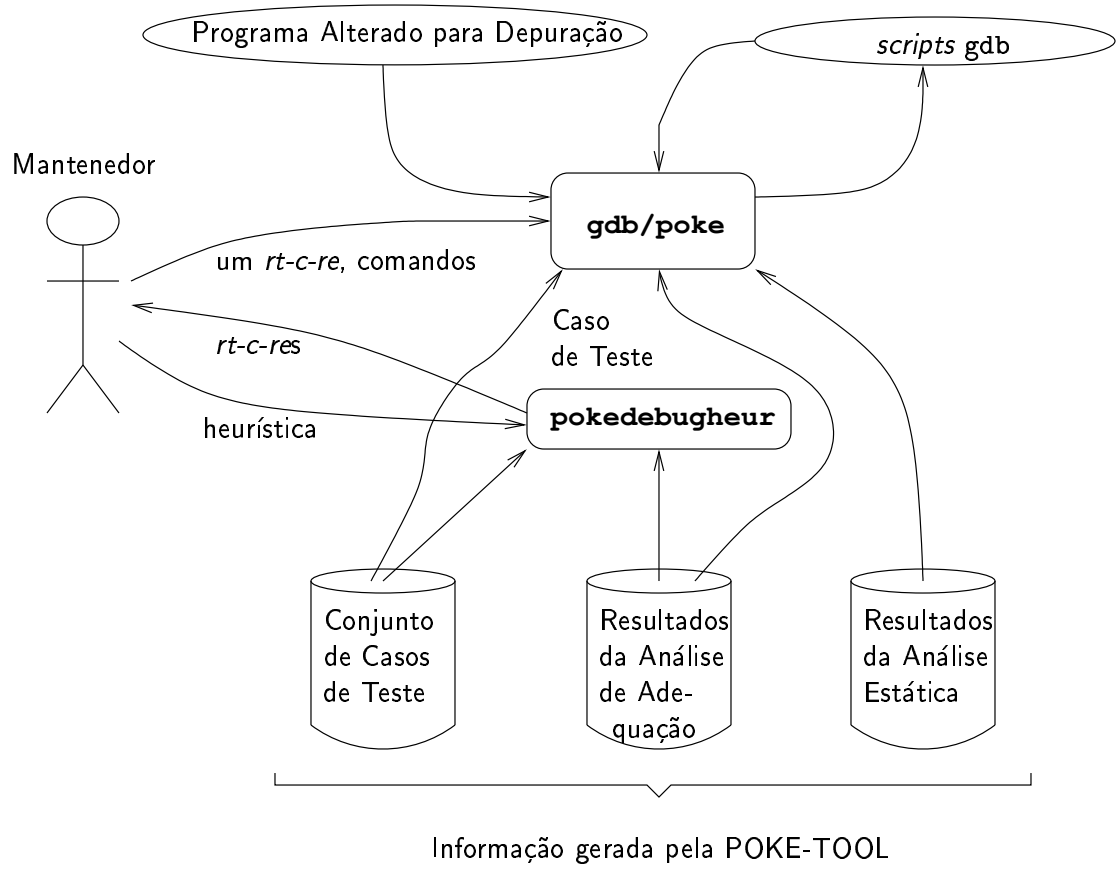


Figura 6.1: Relacionamento da `gdb/poke` com a POKE-TOOL e o módulo `pokedebugheur`.

para depuração utilizando-se o compilador gcc com a opção de depuração -ggdb. Este procedimento é necessário para incluir as informações exigidas pelo depurador simbólico gdb.

O programa alterado para depuração utiliza código adicional para registrar o nó corrente e a sua posição na trajetória. A Figura 6.2 contém um trecho da versão para depuração do programa Sort. A instrução macro ponta_de_prova() atribui o *número do nó* corrente e a sua *posição* nas variáveis locais num_no e seq_exec, respectivamente. Assim, em qualquer ponto da execução, é possível identificar o nó em execução e a sua posição na trajetória do caso de teste.

A monitoração de eventos de teste requer ainda que o programa alterado para depuração inclua código adicional para acessar os pontos de *saída* dos nós e para controlar as invocações do procedimento. Pode-se observar na Figura 6.2 que a instrução macro stop é um comando não-operacional cujo objetivo é *marcar* pontos especiais do programa. O primeiro stop determina o ponto de entrada do procedimento de maneira que, quando o programa atinge este ponto, a variável num_invoc, responsável por indicar a invocação ativa do procedimento, já está ajustada com o número da invocação corrente. Os demais stops, por sua vez, estabelecem os pontos de *saída* dos nós. Assim, quando o programa pára nestes pontos, todas as definições de variáveis do nó já ocorreram e os novos valores podem ser observados. Note-se, entretanto, que os pontos de *entrada* não necessitam de comandos especiais para marcá-los, pois estão localizados no primeiro comando pertencente aos nós, isto é, imediatamente depois da instrução macro ponta_de_prova(). Nestes pontos, a ponta_de_prova() já foi executada e, portanto, as variáveis num_no e seq_exec estão com os valores ajustados.

Alguns comandos de controle de fluxo requerem tratamento especial quando eventos de teste são monitorados. Por exemplo, os comandos associados à iniciação e ao incremento do comando for são retirados do corpo desse comando. Este artifício é necessário porque, caso contrário, seria impossível parar a execução do programa na *saída* dos nós associados à iniciação e ao incremento do for. Já os comandos de seleção condicional (e.g., if e switch) requerem que instruções macros stop sejam inseridas em cada possível ponto de saída do comando. No exemplo da Figura 6.2, os comandos de iniciação e o incremento do for foram retirados do corpo do comando e instruções macros stop foram inseridas depois de cada possível saída do comando if.

6.3 Acessando Eventos de Teste

6.3.1 Requisitos de Teste de Unidade

Os eventos associados a um requisito de teste são acessados na gdb/poke utilizando os comandos first_def, instance_def, last_def, next_use (next_puse), instance_use (instance_puse) e next_check. Estes comandos são implementados utilizando *scripts* escritos na linguagem de programação do depurador gdb. Os *scripts* são específicos ao particular requisito de teste escolhido para ser monitorado; eles são gerados e carregados sempre que o mantenedor executa o comando breakassoc.

```
#define ponta_de_prova(num) (num_no=num,++seq_exec)
#define stop if(0)
...
cmp(i, j)
char *i, *j;
/*  1 */        {
                   static long num_invoc = 0; long seq_exec=0; int num_no = 0;
...
                   num_invoc++;
                   stop;             /* ponto de entrada da função */
                   ponta_de_prova(1);
/*  1 */           k = nfields>0;    /* ponto de entrada nó 1 */
                   stop;             /* ponto de saída nó 1 */
                   ponta_de_prova(2);
/*  1  2 78 */     for(;k<=nfields;) /* ponto de entrada nó 2 */
/*  3 */           {
                      stop;          /* ponto de entrada nó 2 */
                      ponta_de_prova(3);
...
/*  3 */              if(k)
/*  4 */              {
                         stop;       /* ponto de saída nó 3 */
                         ponta_de_prova(4);
...
/*  4 */              }
/*  5 */              else
/*  5 */              {
                         stop;       /* ponto de saída nó 3 */
                         ponta_de_prova(5);
...
/*  5 */              }
...
                      ponta_de_prova(78);
/* 78 */              k++;           /* ponto de entrada nó 78 */
                      stop;          /* ponto de saída nó 78 */
                      ponta_de_prova(2);
/* 78 */           }
...
/* 82 */        }
...
```

Figura 6.2: Fragmento da versão alterada para depuração do programa `Sort`.

Assim, toda vez que o comando `breakassoc` é executado, os *scripts* anteriores são eliminados e novos *scripts*, relativos ao novo requisito a ser investigado, são carregados. O funcionamento do monitoramento de eventos usando *scripts* do `gdb` é descrito a seguir.

Durante a geração da versão alterada para depuração é criado um arquivo no qual estão indicados os pontos (linhas do programa alterado) de entrada e saída de cada nó. Assim, para monitorar os eventos de um requisito, utiliza-se esta informação para colocar *breakpoints* em pontos estratégicos do programa, a saber, na saída do nó onde ocorre a definição das variáveis (nó de definição), na saída dos nós onde ocorre a redefinição de alguma dessas mesmas variáveis (nó de redefinição) e na entrada do nó onde elas são (potencialmente) usadas (nó de uso).

Assim, sempre que o mantenedor executa os comandos `next_use` (`next_puse`), `instance_use` (`instance_puse`) ou `instance_def` o programa é executado e os eventos de teste são verificados durante a execução. Considere-se, por exemplo, o comando `next_use`. Este comando executa o programa e verifica os nós atingidos em cada *breakpoint* encontrado. Toda vez que é atingido o nó de definição é indicado que a *condição* para ocorrência do evento desejado (no caso, alcance do uso) está válida. Por outro lado, se um nó de redefinição é atingido, esta indicação é retirada. Finalmente, quando é atingida a entrada do nó de uso, a validade da condição do evento é verificada; se válida, o evento se completou e a execução do programa é interrompida; caso contrário, a execução continua. Os demais comandos referenciados acima funcionam de maneira similar.

Os comandos `first_def` e `last_def` não determinam os eventos em tempo de execução. Ao contrário, para acessar os eventos, eles utilizam informação coletada durante a análise de adequação. Esta informação consiste nas instâncias do nó de definição na primeira e na última instância do requisito. Na `gdb/poke`, a instância de um nó é definida pelo número do nó, pela posição na trajetória e pelo número da invocação do procedimento. Assim, os comandos `first_def` e `last_def` alcançam diretamente os eventos simplesmente executando o programa e verificando, cada vez que o nó de definição é atingido, se a posição na trajetória e o número de invocação são iguais ao da instância procurada.

O comando `next_check`, por sua vez, realiza as mesmas funções do comando `next` do depurador `gdb`, porém, verificando a ocorrência de eventos de teste. Ele funciona de maneira semelhante ao comando `next_use`, mas não utiliza *breakpoints*. Como o comando `next_check` executa o programa passo a passo, a ocorrência dos eventos é verificada cada vez que um novo nó é atingido. Caso um evento tenha ocorrido, uma mensagem é emitida com os valores dos dados na entrada ou na saída do novo nó, dependendo de onde está localizado o ponto de parada do evento.

No Apêndice D, são apresentados todos os *scripts* gerados pela `gdb/poke` para a *adu* `(65,(68,69),pa)` utilizados na segunda sessão de depuração apresentada no Capítulo 5.

6.3.2 Requisitos de Teste de Integração

A versão atual da gdb/poke não monitora eventos associados aos requisitos de teste de integração; todavia, as idéias desenvolvidas nos *scripts* discutidos na subseção anterior podem ser utilizadas, com poucas modificações, para a monitoração de eventos de teste de integração.

Os *scripts* dos comandos `instance_def`, `next_use` (`next_puse`), `instance_use` ou (`instance_puse`) requerem poucas modificações para serem usados no contexto de integração. Os *breakpoints* continuam a ser colocados nas saídas dos nós de definição e redefinição e na entrada do nó de uso. A diferença, porém, é que estes nós ocorrem em procedimentos distintos. Este fato requer que os nós alcançados durante a execução do programa sejam identificados pelo número e pelo nome do procedimento onde ocorrem, pois somente o número do nó não os identifica univocamente. Os *scripts* dos comandos acima podem ser alterados para obter o nome do procedimento corrente visto que a pilha de execução, acessível pelo gdb, contém esta informação. Analogamente, os comandos `first_def` e `last_def` pode ser implementados neste contexto simplesmente adicionando o controle do procedimento corrente.

O comando `next_check`, entretanto, é o que requer maiores modificações para tratar os eventos dos requisitos de teste de integração. Este comando, no teste de unidade, executa passo a passo os comandos do mesmo procedimento sem *visitar* os procedimentos invocados. Neste contexto, isto é suficiente, pois somente os requisitos internos à unidade estão sendo analisados. No entanto, quando são monitorados os eventos dos requisitos de integração, a execução passo a passo não pode ficar restrita a um procedimento; ela deve envolver os dois procedimentos *alvos* relacionados com o requisito de integração. Para que o comando `next_check` execute passo a passo os trechos de código envolvendo esses dois procedimentos, é necessário redefini-lo utilizando os comandos `step` e `finish` do gdb.

O comando `step` executa passo a passo o programa, *visitando* inclusive os procedimentos invocados. No entanto, o procedimento visitado pode não ser um dos dois envolvidos no requisito de integração. Neste caso, a execução passo a passo deve ser retomada nos procedimentos alvo. Isto pode ser realizado executando-se o comando `finish` toda vez que o procedimento visitado é diferente de um dos procedimentos envolvidos no requisito de integração. O comando `finish` executa o procedimento corrente até o final e retorna ao procedimento anterior para execução passo a passo. De maneira similar ao comando `next_check` para eventos de teste de unidade, a ocorrência de eventos deve ser verificada a cada novo nó visitado nos procedimentos alvo.

6.4 Selecionando Requisitos Candidatos a Revelar Erros

Os novos requisitos a serem investigados são selecionados a partir de consultas aos resultados da análise de adequação do caso de teste usado na sessão de depuração. Para realizar essas consultas em tempo de depuração, a gdb/poke carrega os resultados do procedimento em análise quando o comando

`breakassoc` é executado. No entanto, isto ocorre somente na primeira vez em que um requisito do procedimento é monitorado. Os dados propriamente ditos da análise de adequação são armazenados em um vetor (chamado *vetor de informações de requisitos*) cuja estrutura é descrita a seguir:

1. conjunto de variáveis do requisito;
2. nó de definição;
3. nó ou ramo de uso;
4. número de instâncias do requisito no caso de teste;
5. instância do nó de definição na primeira instância do requisito;
6. instância do nó de definição na última instância do requisito.

A operação básica no processo de seleção de requisitos é *consultar* o vetor de informações para determinar o conjunto de requisitos a serem investigados. A *chave de busca* da consulta é composta de um par (nó ou ramo, conjunto de variáveis), sendo que o conjunto de variáveis pode ser vazio. A `gdb/poke` na versão atual implementa uma busca sequencial ao vetor de informações cujo custo é $O(|RT_C|)$. A busca tem este custo porque todos os requisitos exigidos pelo critério C são consultados. No entanto, os requisitos de teste podem ser indexados em função dos nós e dos ramos, bem como pelas variáveis envolvidas neles, de forma que a consulta pode ser indexada e constante para *um nó* ou *ramo* e *uma variável*. Como essa busca indexada deverá ser realizada para *cada* variável, o seu custo é dado por $O(|V|)$, onde V é conjunto de variáveis que possuem valores incorretos.

A seleção **refine_use**, quando usada para identificar requisitos candidatos a revelar erros dos critérios todos usos ou todos p-usos, requer unicamente a operação de consulta ao vetor de informações de requisitos. Nestes casos, o mantenedor deve fornecer à `gdb/poke` o nó ou o ramo, bem como o conjunto de variáveis cujos valores estão incorretos. Portanto, a chave de busca está completa e o custo da seleção **refine_use** aplicada em requisitos dos critérios da família Fluxo de Dados é $O(|V|)$.

Já o critério todos potenciais usos requer que, antes de ser realizada a consulta ao vetor de informações, sejam determinados os ramos essenciais estendidos que alcançam imediatamente o nó onde o conjunto de variáveis com valores incorretos foi detectado. Este conjunto de ramos é determinado pelo algoritmo descrito a seguir.

Para a descrição do algoritmo (escrito em uma linguagem no estilo Pascal), supõe-se que o procedimento sendo investigado possui um grafo de fluxo de controle $G(N,R,e,s)$ e o mantenedor tenha observado que o conjunto de variáveis V possui valores incorretos na instância j^p do nó $j \in N$.

procedure RamosEssenciaisEstendidosImediatos (**var** ListaRamos, **integer** j, **integer** NoOriginal)
begin
for todo ramo $(k, j) \in R$ **do**
  if (k,j) é um ramo essencial estendido **then** ListaRamos $\leftarrow$ (k,j);
  else
    if k é dominador de NoOriginal **then** RamosEssenciaisEstendidosImediatos(ListaRamos,k,j);
done
end

O algoritmo acima é invocado pela primeira vez da seguinte forma: *RamosEssenciaisEstendidosImediatos(ListaRamos, j, j)*. Sua função básica é realizar uma busca em reverso e em profundidade no grafo G visitando os ramos que fazem o nó j ser alcançado a partir do nó de entrada e. Para garantir que somente esses ramos sejam visitados, a busca em profundidade é restrita aos ramos cujos nós são *dominadores* de j. Os primeiros ramos essenciais estendidos visitados são então salvos.

O conjunto de nós *dominadores* e o conjunto de *ramos essenciais estendidos* foram determinados durante a análise estática do procedimento para determinar os requisitos de teste (Chaim, 1991); portanto, o custo para determiná-los já foi amortizado durante o teste. Assim, o custo do algoritmo em tempo de depuração é determinado pela busca em profundidade. Se todos os ramos do grafo G forem visitados, este custo é $O(|R|)$ (Hecht, 1977). No entanto, somente os ramos essenciais estendidos que *imediatamente* alcançam j interessam; por isso, o algoritmo não precisa visitar todos os ramos do grafo. Logo, o custo acima é o de *pior caso*.

Portanto, a chave de busca para determinar as *adpu*s candidatas a revelar erros é composta dos ramos essenciais estendidos determinados pelo algoritmo acima e o conjunto de variáveis V. O custo total para determinar as *adpu*s utilizando a seleção **refine_use** é definido pelo custo do algoritmo *RamosEssenciaisEstendidosImediatos* ($O(|R|)$) mais o custo da consulta ao vetor de informações para cada variável e ramo essencial estendido determinado ($O(|V| \times |R|)$). Assim, o custo total é $O(|V| \times |R|)$ no *pior caso*.

A seleção **refine_branch** requer também a inspeção do grafo de fluxo de controle para determinar a chave de busca. No caso, trata-se dos ramos que englobam imediatamente o nó j cuja instância j^p foi incorretamente alcançada. Utilizando esses ramos, o vetor de informações é consultado para identificar os requisitos a serem investigados. O algoritmo a seguir seleciona os ramos que englobam imediatamente o nó j.

procedure RamosEnglobandoImediatamente (**var** ListaRamos, **integer** j)
begin
for todo ramo (k, j) $\in R$ **do**
  if suc(k) > 1 **then** ListaRamos $\leftarrow$ (k,j);
  else
    if suc(k) $= 1$ **and** k é dominador de j
    then RamosEnglobandoImediatamente (ListaRamos,k);
done
end

A função *suc(k)* retorna o número de sucessores de k. O algoritmo acima é análogo ao algoritmo anterior e também visita os ramos que fazem com que o nó j seja alcançado a partir do nó de entrada. Os primeiros ramos que englobam j são selecionados para a consulta ao vetor de informações. Por se tratar de uma busca em profundidade, o custo do algoritmo no pior caso também é $O(|R|)$. O custo total da seleção **refine_branch** é definido pelo custo do algoritmo *RamosEnglobandoImediatamente* mais o custo da consulta ao vetor de informações para todos os ramos selecionados. No entanto, para **refine_branch**, o custo da consulta ao vetor de informações para um único ramo é constante, pois o conjunto de variáveis V é vazio. Supondo o pior caso em que todos os ramos são selecionados, o custo total da consulta é $O(|R|)$. Assim, a complexidade de **refine_branch** é também limitada a $O(|R|)$.

A complexidade espacial do mecanismo de seleção de requisitos de teste candidatos a revelar erros é *constante* visto que todas as estruturas de dados (e.g., grafos de fluxo de controle, vetor de informações de requisitos) são estáticas. Note-se que, ao contrário de outras ferramentas (PELAS (Korel, 1988; Korel e Rilling, 1997, 1998), FIND (Shimomura, 1993; Shimomura et al., 1995) e Spyder (Agrawal, 1991; Viravan, 1994)), não há restrições de memória para utilização das seleções **refine_use** e **refine_branch**.

Do ponto de vista da complexidade temporal, o mecanismo de seleção de requisitos, no pior caso, possui complexidade $O(|V| \times |R|)$, sendo que este custo ocorre toda vez que ele é invocado. A alternativa seria utilizar algoritmos para determinação de *fatias* dinâmicas sem restrições de memória (Korel e Yalamanchili, 1994; Gyimóthy et al., 1999). No entanto, o custo desses algoritmos é proporcional ao número de comandos executados até o ponto de execução onde se deseja determinar a *fatia*. Como o processo de depuração requer a determinação sucessiva de vários *fatias* dinâmicas, o custo desses algoritmos se torna inviável para programas em que o número de comandos executados é grande (Nishimatsu et al., 1999).

Portanto, o custo das seleções **refine_use** e **refine_branch**, tanto em termos de memória como de tempo, é bem menor que as possíveis alternativas. Porém, deve-se observar que o custo menor é obtido às custas da precisão da informação. A informação fornecida pelo mecanismo de seleção de requisitos não tem a precisão de uma *fatia* dinâmica; porém, o seu custo em tempo de depuração é inferior.

6.5 Alterações no Depurador gdb

A implementação da gdb/poke exigiu poucas modificações no código original do depurador gdb. Os únicos comandos adicionados ao conjunto de comandos do gdb foram `breakassoc`, `refine_use` e `refine_branch`; os demais são *scripts* gerados cada vez que é executado o comando `breakassoc`. Além disso, estes três comandos basicamente utilizam informações geradas pela POKE-TOOL durante o teste e não interagem com o código do gdb propriamente dito.

6.6 Limitações

Os comandos `first_def` e `last_def` acessam facilmente os eventos de teste relacionados a eles porque as instâncias dos nós de definição na primeira e última instância do requisito de teste são armazenadas durante a análise de adequação do caso de teste. O ideal seria armazenar este tipo de informação para todas as instâncias do requisito de teste. No entanto, dado o grande número de instâncias que ocorrem durante a execução de um caso de teste, isto não é possível.

Por isso, as demais instâncias dos eventos são determinadas em tempo de execução. A desvantagem dessa solução é que, para atingir a *definição* de uma instância em particular, é necessário reexecutar o programa. Para exemplificar essa situação, o segundo exemplo do Capítulo 5 é retomado. Digamos que o mantenedor queira verificar a definição da instância de número treze da *adu* (65,(68,69), pa) cujo p-uso acabou de ser alcançado. Para atingir o evento desejado, é necessário executar os comandos a seguir:

```
2382    /* 65 */                pa++;
Def node has been hit at node 65 (65,188,23)
Variable pa value: 0x805d33e ""
2398    /* 68 */              if(pa>=la || *pa=='\n')
2402    /* 69 */                 if(pb<lb && *pb!='\n')
P-use node has been reached at node 69 (69,188,27)
Variable pa value: 0x805d33e ""
Instance # 13
Use instance 13 has been reached
(gdb/poke) rerun
Entered function cmp -- 5th invocation
(gdb/poke) instance_def
2382    /* 65 */                pa++;
Def node has been hit at node 65 (65,188,23)
Variable pa value: 0x805d33e ""
```

A execução do comando `instance_def`, depois de reexecutado o programa, faz com que ele pare na definição associada à última instância visitada. Isto é possível porque durante a execução anterior foram

armazenados o número da invocação e a posição na trajetória relativos à definição da última instância visitada. Neste sentido, o comando `instance_def` funciona da mesma maneira que os comandos `first_def` e `last_def`, porém, exigindo uma execução prévia.

Outra limitação ocorre quando os requisitos de teste selecionados por **refine_use** e **refine_branch** são investigados. As instâncias desses requisitos que devem ser examinadas são aquelas que estão *encadeadas* com a instância do requisito que revelou um erro. A versão atual da `gdb/poke` não apóia este encadeamento de instâncias de requisitos; entretanto, ela o simula em parte permitindo que somente eventos de teste restritos a uma particular invocação do procedimento sejam examinados. O comando `next_use_invoc` (`next_puse_invoc`) pára apenas em eventos de teste que pertencem a uma invocação particular do procedimento cujo número de invocação é passado como parâmetro.

Este comando é particularmente útil quando os novos requisitos a serem investigados pertencem a um outro procedimento como, por exemplo, nas situações do tipo *Valor de Retorno Incorreto*. Nestas situações, a investigação para identificação dos requisitos selecionados pode ficar restrita à particular invocação da função que retornou um valor incorreto.

Na verdade, as limitações descritas acima não estão relacionadas com a `gdb/poke` em si, mas com o fato dos depuradores disponíveis na prática não permitirem execução em reverso. Os dois problemas poderiam ser elegantemente solucionados utilizando execução em reverso. Por exemplo, para atingir a definição de uma instância, depois de ter sido atingido o uso, bastaria executar o programa em reverso até o próximo *breakpoint*, que está localizado no nó de definição. Entretanto, a investigação dos novos requisitos a serem investigados é que se beneficiaria mais com a disponibilidade dessa função. A `gdb/poke` poderia colocar *breakpoints* em todos os nós de definição dos requisitos selecionados e executar em reverso o programa até que um desses *breakpoints* fosse atingido. Assim, as instâncias dos requisitos candidatos a revelar erros que encadeiam com a instância do requisito que revelou o erro poderiam ser investigadas.

Como discutido no Capítulo 2, a execução em reverso tem sido pesquisada durante muito tempo; porém, não temos conhecimento de depuradores comerciais com esta função. No entanto, recentemente, algoritmos eficientes para execução *biderecional* de programas foram propostos (Boothe, 2000). Como prosseguimento desse trabalho, pretende-se investigar a escalabilidade desses algoritmos para programas reais e, caso os resultados sejam promissores, gerar uma nova versão da `gdb/poke` que permita execução em reverso.

6.7 Considerações Finais

Neste capítulo, foi apresentada a implementação da ferramenta `gdb/poke`. Ela é uma extensão do depurador simbólico `gdb` que implementa os mecanismos de monitoração de eventos e de seleção de novos requisitos candidatos a revelar erros, apoiando, dessa maneira, a aplicação da estratégia **DRT** introduzida no Capítulo 5. A característica mais importante da implementação realizada é que ela utiliza

os recursos comuns do depurador gdb e algoritmos de ordem linear. Por isso, a ferramenta não possui as restrições existentes em ferramentas similares, o que poderá torná-la uma alternativa para uso em aplicações reais.

Capítulo 7

Conclusões

7.1 Síntese do Trabalho

Neste trabalho foi investigado o uso na depuração de software de informação coletada durante a fase de teste. O objetivo foi avaliar a conjectura de que a *utilização* de informação de teste na depuração pode reduzir os custos de aplicação do teste sistemático e das técnicas de depuração mais modernas em sistemas reais. Por estar centrada no uso de informação de teste, esta investigação voltou-se para a depuração que ocorre *depois* dessa atividade. Este tipo de depuração envolve o software com todas as funções definidas na especificação já incluídas e é uma decorrência do teste bem sucedido. Se o teste conduzido foi sistemático, então os resultados desta fase podem ser utilizados na depuração.

7.1.1 Técnicas de Depuração e o Paradigma DDT

Inicialmente foi analisado o estado da arte das técnicas de depuração. Para esta análise utilizou-se como guia o paradigma de *Depuração depois do Teste* (DDT). Este paradigma tem como aspectos principais a identificação, a avaliação e o refinamento sucessivo de sintomas internos até a localização do defeito e o tipo de informação de teste utilizada. Além disso, as técnicas de depuração foram revisadas com relação a sua escalabilidade para sistemas reais. O resultado da análise mostrou que as técnicas de depuração são mais promissoras do ponto de vista de escalabilidade quando utilizam informação de teste. Isto ocorre porque seu custo principal já foi amortizado durante o teste. No entanto, as técnicas que utilizam esse tipo de informação apóiam apenas um dos passos do paradigma DDT — a *identificação* de possíveis sintomas internos. Os demais passos — *avaliação* e *refinamento* (seleção) — não são apoiados.

7.1.2 Teste de Fluxo de Dados de Programas com Ponteiros e Campos de Registros

Uma questão importante investigada nesta tese foi o uso de informação de fluxo de dados mais precisa no teste de programas que utilizam ponteiros e campos de registros. Do ponto de vista da

depuração depois do teste, a disponibilidade dessa informação mais precisa é interessante, pois ela fornece detalhes sobre o uso de ponteiros e registros que podem facilitar a localização dos defeitos. A investigação baseou-se na definição e implementação de dois modelos de dados (chamados modelos de dados **nível 1** e **nível 2**) baseados na abordagem conservadora para o teste de programas que utilizam estes recursos (Vilela et al., 1997). Eles estabelecem níveis crescentes de precisão da análise de fluxo de dados utilizada para determinar os requisitos dos critérios das famílias Fluxo de Dados e Potenciais Usos. A conjectura é que a análise de fluxo de dados mais precisa aumenta a eficácia a um custo razoável (Marx e Frankl, 1996, 1999).

Os modelos propostos foram comparados por meio de um estudo de caso com um modelo (chamado **nível 0**) que não leva em consideração os fluxos relativos a ponteiros e campos de registros. Neste estudo de caso, a *eficácia* e o *custo* do teste de fluxo de dados dos critérios das famílias Fluxo de Dados e Potenciais Usos, utilizando os modelos **níveis 0**, **1** e **2**, foram examinados com respeito a várias versões defeituosas do programa `Sort` do ambiente Unix. As versões utilizadas continham defeitos relativos ao uso indevido de ponteiros e campos de registros. As principais observações do estudo de caso são descritas abaixo.

Para as versões defeituosas do programa `Sort` (padrão do ambiente Unix) utilizadas no estudo de caso, observou-se que, para os critérios da família Fluxo de Dados, a *eficácia* e o *tamanho* dos conjuntos de casos de teste adequados aos critérios permaneceram praticamente constantes quando foram utilizados os modelos de dados mais precisos. A razão desse comportamento deve-se aos fluxos de controle e de dados particulares ao programa utilizado no experimento. Já para os critérios Potenciais Usos, o *tamanho* dos conjuntos adequados aumentou quando os modelos de dados mais precisos foram utilizados. Isto significa que as *adpu*s adicionais requeridas pelos modelos mais precisos exigem novos caminhos a serem testados. Porém, em termos de *eficácia*, apenas uma das versões defeituosas apresentou um aumento marcante quando os novos modelos foram utilizados. A análise dos resultados desta versão mostrou que o aumento de eficácia foi causado por *adpu*s — determinadas unicamente pelos modelos mais precisos — que exigem caminhos com grande probabilidade de revelar o defeito.

Portanto, os resultados com os critérios Potenciais Usos indicam que há defeitos cuja detecção é facilitada quando modelos de dados mais precisos são utilizados. No caso dessa família de critérios, o custo adicional dos modelos mais exigentes parece ser compensador visto que o aumento no número de novos requisitos de teste (22,85%) e no de casos de teste (12%) é razoável. Isto se deve à utilização de *adpu*s baseadas em ramos essenciais estendidos que reduz o número de requisitos de teste dos critérios Potenciais Usos.

No entanto, para os critérios da família Fluxo de Dados, o custo para obter esta informação detalhada cresceu consideravelmente (aumento de 90% no número de requisitos de teste quando é utilizado o **nível 2**). Portanto, a utilização desses critérios com modelos de dados mais precisos requer que mecanismos de redução do número de requisitos de teste sejam utilizados. Uma possível solução seria usar o algoritmo de inclusão estática de associações de Marré e Bertolino (1996).

7.1.3 Uso de Requisitos de Teste na Localização de Defeitos

O uso de informação de teste propriamente dita na depuração foi investigado por meio de um estudo empírico. Este estudo investigou a habilidade dos requisitos de teste em identificar informação útil para a depuração utilizando heurísticas. O objetivo foi avaliar a habilidade de uma heurística H utilizando os requisitos de teste de um critério C em identificar informação útil para a localização do defeito. Esta informação pode ser um trecho de código onde o defeito está contido ou um subconjunto de requisitos que fornecem indicações úteis, obtidas em tempo de execução, para a sua localização. Para capturar essas indicações em tempo de execução, foi proposto o conceito de *requisito de teste revelador de erro*.

O estudo empírico consistiu na aplicação de diferentes pares (H, C) em vários conjuntos de casos de teste gerados segundo três cenários de teste-depuração (GT-C, TP-C e MT-C) para localizar defeitos em versões incorretas do programa `Sort`. Os cenários foram definidos para simular diferentes contextos de teste e depuraçao: o cenário GT-C simula organizações onde há um grupo independente de teste que desenvolve conjuntos adequados a um critério C; o cenário TP-C representa aquelas situações em que o testador desenvolve um caso de teste por vez para satisfazer o critério C e interrompe o teste quando um defeito é encontrado; e o cenário MT-C modela a depuração que ocorre durante a manutenção utilizando os resultados do teste com o critério C. Os dados obtidos foram comparados considerando-se aspectos de *inclusão*, *eficiência* e *custo*. O objetivo foi investigar o papel desempenhado pela heurística e pelo requisito de teste utilizados, bem como do cenário de teste-depuração, na habilidade do par (H, C) em identificar informação útil para a depuração.

Com relação ao tipo de heurística utilizado, os resultados confirmaram indicações de estudos anteriores (Collofello e Cousins, 1987; Pan e Spafford, 1992) de que as heurísticas do tipo *ranking* são as mais *inclusivas* e *eficientes*. No entanto, elas obtiveram este resultado quando aplicadas nos cenários de teste-depuração GT-C e MT-C. Infelizmente, para o cenário TP-C, que é o mais realista, as heurísticas do tipo *ranking* não obtiveram bons resultados. Por isso, o estudo indica que os pares (heurística do tipo *ranking*, critério de teste) devem ser aplicados em conjuntos de casos de teste que estejam o mais próximo possível daqueles obtidos nos cenários GT-C e MT-C.

Para o programa do estudo de caso, os requisitos dos critérios de fluxo de dados foram os que obtiveram os melhores resultados em termos de *inclusão* e *eficiência* utilizando as heurísticas do tipo *ranking* nos cenários mais favoráveis. No entanto, quando comparados com a utilização das mesmas heurísticas com requisitos de critérios de fluxo de controle nas mesmas condições, observou-se que as medidas de *inclusão* e *eficiência* não são marcadamente superiores, embora o *custo* de se utilizar os requisitos de critérios de fluxo de dados seja bem maior.

O resultado mais interessante deste estudo, porém, foi a observação de que os pares (H,C) mais promissores são mais inclusivos e eficientes na seleção de requisitos reveladores de erros do que na seleção de um trecho de código que contém o sítio do defeito. Esta observação motivou o desenvolvimento da *estratégia de* **D***epuração baseada em* **R***equisitos de* **T***este* (**DRT**). O objetivo desta estratégia é localizar

o sítio do defeito pela identificação sucessiva de requisitos de teste reveladores de erros.

7.1.4 DRT - Uma Estratégia de Depuração baseada em Requisitos de Teste

A estratégia **DRT** é baseada no uso de heurísticas para determinar um conjunto inicial de *requisitos candidatos a revelar erros* e em dois mecanismos — monitoramento de eventos de teste e seleção de novos requisitos — cujo objetivo é apoiar a identificação sucessiva de requisitos reveladores de erros até a localização do sítio do defeito.

O primeiro mecanismo possibilita que o mantenedor acesse os eventos e as trajetórias relacionados com as instâncias de um requisito de teste. Monitorando os eventos de teste, ele pode investigar se o requisito inclui o sítio do defeito no trecho de código associado ou se ele fornece apenas indicações úteis para a depuração. Dessa análise, o mantenedor pode concluir ainda que o requisito não contribui com nenhuma informação relevante para a localização do defeito. O mecanismo utiliza os resultados da análise de adequação para monitorar os eventos de teste.

O segundo mecanismo auxilia a identificação de novos requisitos de teste a serem investigados quando o requisito de teste revelador de erro fornece apenas indicações para a localização do defeito, mas não o seu sítio. Ele indica requisitos que podem afetar o requisito de teste revelador de erro e que foram exercitados pelo caso de teste. O mecanismo, quando aplicado nas situações adequadas, move o processo de localização adiante visto que o conjunto de requisitos selecionados inclui pelo menos um requisito de teste revelador de erro. Foi mostrado que a identificação sucessiva dos requisitos reveladores de erro presentes nos conjuntos identificados pelo segundo mecanismo termina por indicar um requisito tal que o sítio do defeito está no trecho de código do requisito ou está associado à trajetória da instância que revela o erro.

Para a utilização efetiva dessa estratégia foram desenvolvidas duas ferramentas: `pokedebugheur` e `gdb/poke`. A `pokedebugheur` implementa as heurísticas propostas na literatura para localizar defeitos utilizando requisitos de teste. Já a `gdb/poke` implementa os mecanismos de monitoração de eventos e de seleção de novos requisitos de teste a serem investigados. Ambas as ferramentas têm um custo baixo em tempo de depuração, pois o custo principal ocorre durante a análise de adequação dos casos de teste. A implementação da `gdb/poke`, em especial, é baseada na utilização dos recursos comuns do depurador simbólico `gdb` e de algoritmos de ordem linear. Por isso, a estratégia apoiada por estas ferramentas possui baixo custo em tempo de depuração em comparação com estratégias similares, constituindo-se em uma possível alternativa para uso em programas reais.

Assim, o principal resultado dessa tese é densenvolvimento de uma estratégia de depuração guiada pelos resultados da atividade de teste e que apóia todos os passos paradigma DDT. Na estratégia **DRT**, o uso de heurísticas baseadas em requisitos de tese realiza o *mapeamento dos resultados do teste em possíveis sintomas internos* (passo 2). O monitoramento de eventos, por sua vez, apóia a *avaliação dos possíveis sintomas internos* (passo 3) utilizando os resultados da análise de adequação, enquanto

o mecanismo de seleção de requisitos identifica *novos possíveis sintomas internos* (passo 4) entre os requisitos exercitados por um caso de teste, movendo o processo de localização em direção ao sítio do defeito. Note-se que o fato de apoiar todas os passos do paradigma DDT representa um salto de qualidade na atividade de depuração visto que o mantenedor não dependerá apenas da sua experiência para selecionar novos possíveis sintomas internos. Além disso, quando os resultados do teste sistemático estão disponíveis, o custo de utilização da estratégia é baixo.

7.1.5 Condições Ideais para Aplicação da Estratégia DRT

As condições ideais para aplicação da estratégia **DRT** ocorrem quando a atividade de teste é sistemática e os conjuntos de casos de teste gerados são adequados a critérios baseados em análise de fluxo de dados. Isto se deve a duas razões: os resultados do estudo empírico indicam que os critérios de teste de fluxo de dados, utilizando heurísticas tipo *ranking*, são os mais inclusivos e eficientes, sendo que isto ocorre no cenário GT-C, aquele em que os conjuntos gerados são adequados aos critérios de teste; além disso, o mecanismo de seleção de novos requisitos, para que possa ser aplicado, requer que os resultados da análise de adequação com relação a critérios de fluxo de dados estejam disponíveis.

No entanto, apesar de as condições ideais para aplicação da estratégia **DRT** ocorrerem quando o teste de fluxo de dados é conduzido, isto não significa que este tipo de teste tenha que ser obrigatoriamente realizado. Na hipótese de somente o teste de ramos ter sido conduzido, o mantenedor pode realizar a análise de adequação com respeito aos critérios de fluxo de dados de alguns casos de teste reveladores de defeito e utilizar as *adu*s e *adpu*s associadas aos ramos identificados utilizando os pares (heurística tipo *ranking*, todos ramos) como os requisitos de teste a serem investigados. Nesta situação, a sessão de depuração deve ser conduzida utilizando os casos de teste para os quais a análise de adequação dos requisitos de fluxo de dados está disponível.

Outra observação importante é que a utilidade da estratégia não está restrita à depuração *depois* do teste. O estudo empírico indica que as heurísticas do tipo *ranking* possuem habilidade para selecionar requisitos de teste reveladores de erros no cenário MT-C. Portanto, a estratégia proposta pode ser também aplicada na depuração *durante* a manutenção.

7.1.6 Validade da Conjectura da Tese

Assim, os resultados obtidos indicam que a informação coletada durante o teste sistemático, em especial do teste estrutural, pode ser usada de maneira eficiente em todas as tarefas de depuração. Dessa maneira, quando isto ocorre, é razoável alocar parte do custo do teste ao de manutenção, mais especificamente, ao custo de depuração. Portanto, a conjectura de que a utilização de informação de teste na depuração pode reduzir os custos de aplicação do teste sistemático e das técnicas de depuração mais modernas em sistemas reais é verdadeira. No caso das técnicas de depuração, não há somente redução de custo; o próprio uso das técnicas pode vir a ser viabilizado quando se utiliza informação de

teste na depuração.

7.2 Contribuições

As principais contribuições desse trabalho descritas na subseção anterior são resumidas abaixo:

1. *Avaliação das técnicas de depuração em relação ao paradigma de Depuração depois do Teste (DDT) e quanto a sua escalabilidade para sistemas reais.*

2. *Definição e implementação de dois modelos de análise de fluxo de dados mais precisos para o teste de programas que utilizam ponteiros e campos de registros e realização de estudo de caso para verificar a eficácia e o custo de utilização desses modelos mais precisos.*

3. *Avaliação do uso de requisitos de teste na depuração por meio de heurísticas e introdução do conceito de requisito de teste revelador de erro.*

4. *Definição de uma* estratégia de **D**epuração baseada em **R**equisitos de **T**este *(***DRT***) que apóia todos os passos do paradigma DDT.*

5. *Implementação de mecanismos eficientes em tempo de depuração para monitoramento de eventos de teste e seleção de novos requisitos a serem investigados na ferramenta* `gdb/poke`.

7.3 Trabalhos Futuros

A partir dos resultados obtidos, é possível vislumbrar alguns trabalhos futuros.

7.3.1 Teste Incremental de Fluxo de Dados

Com relação ao teste de fluxo de dados utilizando os modelos de dados mais precisos, são necessários mais experimentos para confirmar as observações obtidas do experimento inicial realizado nesta tese. Por isso, pretende-se repetir o mesmo experimento com o programa SPACE. Este programa foi desenvolvido para calcular e fornecer os parâmetros de entrada a um outro software utilizado para encontrar a melhor disposição física de antenas utilizadas em aplicações espaciais. Trata-se de um programa de médio porte (10.000 linhas de código) cujos defeitos *reais*, que ocorreram durante o desenvolvimento do programa, foram catalogados. Além disso, ele possui uma massa de casos de teste de 10.000 elementos gerados aleatoriamente. Devido a essas características, o programa SPACE tem sido utilizado em vários experimentos em teste de software (Crespo et al., 2000; Delamaro et al., 2001b; Frankl e Iakounenko, 1998; Pasquini et al., 1996).

Outra linha interessante de trabalho é investigar o uso de níveis intermediários de análise de fluxo de dados. Observando a Tabela 3.3 nota-se que há um aumento significativo no número de requisitos de

teste exigidos (maior que 330%) quando se decide utilizar o critério todos p-usos com o modelo **nível 0** ao invés do critério todos ramos. Portanto, há um severo aumento de custo quando é tomada esta decisão.

Para dividir o custo do teste de fluxo de dados em passos menores e menos custosos, é possível estabelecer uma estratégia incremental de teste utilizando níveis intermediários de modelos de dados. Por exemplo, ao invés de passar de todos os ramos para todos os p-usos, o teste de fluxo de dados poderia iniciar-se testando apenas os p-usos relativos a variáveis escalares. Caso seja necessário, o escopo do teste pode ser estendido para incluir variáveis agregadas, campos de registros e assim por diante. Note-se que, dependendo das características do programa, a ordem de aplicação dos níveis intermediários de análise de fluxo de dados pode ser diferente. Por exemplo, para programas de estruturas de dados, pode ser mais indicado começar o teste pelas associações relativas ao uso de ponteiros e campos de registros. A estratégia sugerida acima pode ser facilmente implementada em ferramentas como a POKE-TOOL.

7.3.2 Seleção de Informação de Teste para a Depuração

Um dos pontos fundamentais da estratégia **DRT** é a seleção de requisitos de teste candidatos a revelar erros. Portanto, de maneira semelhante, os resultados do estudo da habilidade dos pares (heurística, critério de teste) na seleção de informação útil para a depuração precisam ser confirmados por mais experimentos. O experimento com o programa SPACE pode fornecer dados para a investigação do uso de requisitos de teste na depuração. Porém, outros aspectos devem ser avaliados nesses novos experimentos. Por exemplo, a habilidade dos *requisitos de teste de integração* na localização de defeitos ou o uso de outras técnicas de seleção de requisitos candidatos a revelar erros.

Uma das técnicas a ser investigada é a *Análise de Conceitos* (Ball, 1999). Esta técnica determina pares (T,E) (chamados *conceitos*) tal que T são conjuntos maximais de casos de teste que cobrem determinado subconjunto E de entidades. No contexto de depuração, se o conjunto T de um par (T,E) contém vários casos de teste reveladores de defeitos então as entidades (e.g., requisitos de teste) pertencentes a E são candidatas a serem investigadas. Para a realização dos experimentos com os requisitos de integração, é necessário antes incluir o apoio automatizado ao teste com critérios desse tipo na POKE-TOOL. Por isso, esta ferramenta deve ser estendida para incluir as versões dos critérios das famílias Fluxo de Dados (Jin e Offutt, 1998) e Potenciais Usos (Vilela, 1998) para integração.

Outra linha de pesquisa ainda a ser explorada na seleção de informação de teste para depuração é o uso de requisitos de teste funcionais. Neste trabalho foram investigadas heurísticas baseadas somente nos resultados do teste estrutural; porém, é possível estabelecer, semelhantemente ao que é feito em relação aos requisitos de teste estrutural, um mapeamento entre os requisitos de teste funcional e os casos de teste. Por exemplo, pode-se mapear aos casos de teste as classes de equivalência, os valores limites das classes e os casos de usos (Jacobson et al., 1999) exercitados. Esta informação pode também ser útil para identificar um conjunto de requisitos de teste estruturais candidatos a revelar erros mais

inclusivo e eficiente.

7.3.3 Visualização de Informação de Teste

Na seleção de informação útil para a depuração, um aspecto importante é a visualização da informação de teste. Várias iniciativas têm sido desenvolvidas para auxiliar o mantenedor a visualizar trechos de código das mais variadas maneiras: utilizando cores que indicam a classificação do comando segundo uma heurística tipo *ranking* (Agrawal et al., 1998; Jones et al., 2001) e histogramas que descrevem a execução do caso de teste (Korel e Rilling, 1998). Porém, dois aspectos de visualização precisam ainda ser investigados: a visualização das instâncias dos requisitos de teste (trajetória e valores utilizados); e a visualização dos requisitos de teste estruturais e funcionais nos vários artefatos gerados no processo de software, tais como o diagrama de requisitos (e.g., casos de uso), diagramas de projeto (e.g., diagrama de classes) (Jacobson et al., 1999) e o próprio código fonte. A visualização desses aspectos da informação de teste é importante porque irá auxiliar o mantenedor a selecionar requisitos de teste candidatos a revelar erros e também a determinar aqueles que efetivamente revelam erros.

7.3.4 Ambiente Integrado de Teste e Depuração

A utilização efetiva de informação de teste na depuração requer que ambas as atividade estejam integradas em um único ambiente. Neste ambiente, deve ser possível a manipulação (estática e dinâmica) dos requisitos de teste (estruturais ou funcionais) em qualquer artefato produzido no processo de software. Um passo em direção a esse ambiente é a integração das ferramentas POKE-TOOL e `gdb/poke` em uma interface comum. O passo seguinte é rastrear os requisitos de teste estruturais e funcionais nas representações de software pré-codificação.

7.3.5 Evolução da Atual Implementação da Estratégia DRT

A atual implementação da estratégia **DRT** utiliza as ferramentas `pokedebugheur` e `gdb/poke`. Uma evolução imediata dessa implementação é a introdução de critérios de teste estruturais de integração nas ferramentas. Esta tarefa faz-se necessária não apenas para viabilizar experimentos com requisitos de teste de integração, mas também para tornar a estratégia **DRT** completa.

Especificamente em relação à nova versão da `gdb/poke`, pretende-se incluir a capacidade de execução em reverso utilizando os algoritmos propostos por Boothe (2000). A idéia é verificar a escalabilidade desses algoritmos para programas reais visto que a execução em reverso pode eliminar algumas das limitações atuais da `gdb/poke` e facilitar a identificação de requisitos de teste reveladores de erros.

Outra evolução já mencionada é o desenvolvimento de uma interface comum entre a POKE-TOOL e a `gdb/poke`. Esta ferramenta integrada permitirá a realização de vários experimentos de teste e depuração com o objetivo de avaliar a estratégia **DRT**. Um experimento importante a ser realizado é avaliar

se o baixo custo da estratégia **DRT** é suficiente para torná-la escalável para aplicações reais. Outro experimento relevante é a comparação da depuração apoiada por informação de teste com a depuração *ad hoc* (apoiada somente por depuradores simbólicos) em diferentes cenários de teste-depuração.

Referências Bibliográficas

Adams, E., Muchnick, S. S., 1986. Dbxtool: A window-based symbolic debugger for sun workstation. Software Practice and Experience 16 (7), 653–669.

Agrawal, H., 1991. Towards automatic debugging of computer programs. Tese de Doutorado, Purdue University.

Agrawal, H., Alberi, J. L., Horgan, J. R., Li, J. J., London, S., Wong, W. E., Gosh, S., Wilde, N., 1998. Mining system tests to aid software maintenance. IEEE Computer 31 (7), 64–73.

Agrawal, H., DeMillo, R. A., Spafford, E. H., May 1991. An execution-backtracking approach to debugging. IEEE Software 8 (3), 21–26.

Agrawal, H., Horgan, J. R., 1990. Dynamic program slicing. In: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.

Agrawal, H., Horgan, J. R., London, S., Wong, W. E., 1995. Fault localization using execution slices and dataflow tests. In: Proceedings of the International Symposium on Software Reliability Engineering. IEEE Computer Society Press, Los Alamitos, Calif.

Araki, K., Furukawa, Z., Cheng, J., 1991. A general framework for debugging. IEEE Software Magazine 8 (3), 14–20.

Auguston, M., 1995. A program behavior model based on event grammar and its application for debugging automation. In: Second Workshop on Automated and Algorithmic Debugging. Saint-Malo, France.

Austin, T. M., Breach, S. E., Sohi, G. S., 1994. Efficient dettection of all pointer and array access error. In: Proceedings of the ACM SIGPLAN Conference on Programming Languages Desing and Implementation. Also in *ACM SIGPLAN Notices* 29(6), June 1994.

Ball, T., 1999. The concept of dynamic analysis. In: Proceedings of the 7th European Software Engineering Conference/7th ACM SIGSOFT Symposium on the Foundations of Software Engineering. Toulouse, France.

Balzer, R. M., 1969. Exdams: Extensible debugging and monitoring system. In: Spring Joint Computer Conference. AFIPS Press, Reston, VA, U.S.A.

Boothe, B., 2000. Efficient algorithms for bidirectional debugging. In: ACM SIGPLAN Conference on Programming Language Design and Implementation.

Budd, T. A., 1981. Mutation Analysis: Ideas, Example, Problems and Propects. North-Holland Publishing Company, Cap. Computer Program Testing.

Chaim, M. L., 1991. POKE-TOOL — uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados. Dissertação de Mestrado — FEEC/Unicamp, Campinas.

Chaim, M. L., Maldonado, J. C., Jino, M., Nakagawa, E. Y., Outubro 1998. POKE-TOOL — Estado atual de uma ferramenta para teste estrutural de software baseado em análise de fluxo de dados. In: XII Simpósio Brasileiro de Engenharia de Software, Caderno de Ferramentas.

Chan, T. W., 1997. A framework for debugging. Journal of Computer Information Systems 38 (1), 67–73.

Chen, T. Y., Cheung, Y. Y., 1997. On program dicing. Software Maintenance: Research and Practice 9, 33–46.

Chusho, T., 1987. Test data selection and quality estimation based on the concept of essential branches for program testing. IEEE Transactions on Software Engineering SE-13 (5), 509–517.

Collofello, J. S., Cousins, L., 1987. Toward automatic software fault localization through decision-to-decision path analysis. In: Proceeding of the AFIP 1987 National Computer Conference.

Crespo, A. N., Pasquini, A., Jino, M., Maldonado, J. C., 2000. A binomial software reliability model based on coverage of structural testing criteria. In: 14th Brazilian Simposium on Software Engineering.

Curcio, I. D., 1998. ASAP — a simple assertion pre-processor. ACM SIGPLAN Notices 33 (12), 44–51.

Delamaro, M. E., 1993. PROTEUM — Um Ambiente de Teste Baseado na Análise de Mutantes. Dissertação de Mestrado — ICMSC/USP, São Carlos.

Delamaro, M. E., 1997. Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Mutação. Tese de Doutorado — IFSC/USP, São Carlos, SP, Brasil.

Delamaro, M. E., Maldonado, J., July 1996. Proteum: A tool for the assessment of test adequacy of C programs. In: Conference on Performability in Computing Systems (PCS '96). Brunswick, NJ.

Delamaro, M. E., Maldonado, J. C., Mathur, A. P., March 2001a. Interface mutation: An approach for integration testing. IEEE Transactions on Software Engineering 27 (3).

Delamaro, M. E., Maldonado, J. C., Pasquini, A., Mathur, A. P., 2001b. Interface mutation test adequacy criterion: An empirical evaluation. Empirical Software Engineering 6 (2), 111–142.

DeMillo, R. A., Gwind, D. C., King, K. N., 1988. An extended overview of th Mothra software testing environment. In: Second Workshop on Software Testing, Verification and Analysis. Banff, Canada.

DeMillo, R. A., Lipton, R. J., Sayward, F. G., 1978. Hints on test data selection: Help for the practicing programmer. Computer 11 (4), 34–43.

DeMillo, R. A., Mathur, A. P., 1995. A grammar based fault classification scheme and its application to the classification of the errors of tex. Relatório Técnico SERC-TR-165-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana U.S.A.

DeMillo, R. A., Pan, H., Spafford, E. H., 1996. Critical slicing for software fault localization. In: Proceedings of the International Symposium on Software Testing and Analysis.

Ducassé, M., 1999. Coca: An automated debugger for C. In: Proceedings of the 21st International Conference on Software Engineering. IEEE Computer Society Press, Los Angeles, CA, U.S.A.

Duncan, I., Robson, D., 1996. An exploratory study of common coding faults in C programs. Software Maintenace: Research and Practice 8, 241–256.

Fabbri, S. C. P. F., 1996. A aplicação de análise de mutantes no contexto de sistemas reativos: Uma contribuição para o estabelecimento de estratégias de teste e validação. Tese de Doutorado, IFSC-USP, São Carlos, SP.

Frankl, F. G., Weiss, S. N., Weyuker, E. J., 1985. ASSET—a system to select and evaluate test. In: Proceedings of the IEEE Conference on Software Tools. IEEE Computer Society Press, Los Alamitos, California, U.S.A.

Frankl, F. G., Weyuker, E. J., 1988. An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering 14 (10), 1483–1495.

Frankl, P., Weiss, S., 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. IEEE Transaction on Software Engineering 19 (8), 774–787.

Frankl, P., Weyuker, E. J., 1993. A formal analysis of the fault-detecting of testing methods. IEEE Transactions on Software Engineering 19, 202–213.

Frankl, P. G., Iakounenko, O., 1998. Further empirical studies of test effictiveness. In: Proceedings of the 1998 ACM SIGSOFT Foundations of Software Engineering Conference. Florida, U.S.A.

Frankl, P. G., Weiss, S., Hu, C., 1997. All-uses vs mutation testing: An experimental comparison of effectiveness. Journal of Systems and Software 38, 235–253.

Fritzson, P., Shahmehri, N., Kamkar, M., Gyimothy, T., 1992. Generalized algorithmic debugging and testing. ACM Letters on Programming Languages and Systems 1 (4), 303–322.

Ghezzi, C., Jazayery, M., 1987. Programming Languages Concepts, Second Edição. John Wiley and Sons, New York.

Golan, M., Hanson, D., January 1993. DUEL — A very high-level debugging language. In: Winter USENIX Technical Conference.

Goodenough, J. B., Gerhart, S. L., 1975. Toward a theory of test data selection. IEEE Transactions on Software Engineering SE-2 (3), 156–173.

Gutjahr, W. J., 1999. Partition testing vs. random testing: The influence of uncertainty. IEEE Transactions on Software Engineering 24 (5), 661–674.

Gyimóthy, T., Beszédes, A., Forgács, I., August 1999. An efficient relevant slicing method for debugging. In: SIGSOFT Foundations of Software Engineering. Toulouse.

Harrold, M. J., 2000. Testing: A roadmap. In: Future of Software Engineering, Proceedings of the 22nd International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, California, U.S.A.

Harrold, M. J., Soffa, M. L., March 1991. Selecting and using data for integration testing. IEEE Software 8 (2).

Hastings, R., Joyce, B., 1992. Purify: fast detection of memory leaks and access errors. In: Proceedings of the Winter Usenix Conference.

Hecht, M. S., 1977. Flow analysis of computer programs. Elsevier North-Holland, New York.

Hedley, D., Hennell, M. A., 1985. The causes and effects of infesiable paths in computer programs. In: Proceedings of the 8th International Conference on Software Engineering. United Kingdom.

Herman, P. M., 1976. Data flow approach to program testing. Australian Computer Journal 8 (3), 92–96.

Horgan, J. R., London, S., 1991. Data flow coverage and the C language. In: Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV4). ACM Press, New York, U.S.A.

Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, California, U.S.A.

IEEE, 1991. IEEE Software Engineering Standards Collection. IEEE, New York.

Jacobson, I., Booch, G., Rumbaugh, J., 1999. The Unified Software Development Process. Addison-Wesley, Reading, Massachusetts.

Jin, Z., Offutt, A. J., 1998. Coupling-based criteria for integration testing. Software Testing, Verification, and Reliability 8 (3).

Jones, J. A., Harrold, M. J., Stasko, J. T., 2001. Visualization for fault localization. In: Workshop on Program Visualization at International Conference on Software Engineering.

Kamkar, M., 1995. An overview and comparative classification of program slicing techniques. Journal of Systems and Software 31, 197–214.

Kamkar, M., 1998. Application of program slicing in algorithmic debugging. Information and Software Technology 40, 637–645.

Korel, B., 1988. PELAS — Program Error-Locating Assistant System. IEEE Transactions on Software Engineering 14 (9), 1253–1260.

Korel, B., Laski, J., 1988. Dynamic program slicing. Information Processing Letters 29 (3), 155–163.

Korel, B., Rilling, J., 1997. Application of dynamic slicing in program debugging. In: Third Workshop on Automated and Algorithmic Debugging. Linköping Electronic Articles in Computer and Information Science, Linköping, Sweden, `http://www.ep.liu.se/ea/cis/1997/009/` .

Korel, B., Rilling, J., 1998. Program slicing in understanding large programs. In: International Workshop on Program Comprehension.

Korel, B., Yalamanchili, S., 1994. Forward computation of dynamic program slices. In: International Symposium on Software Testing and Analysis.

Lancevicius, R., Hölzle, U., Singh, A., 1997. Query-based debugging of object-oriented programs. In: OOPSLA Symposium.

Laski, J., Korel, B., 1983. A data flow oriented program testing strategy. IEEE Transactions on Software Engineering SE-9 (3), 347–354.

Lian, L., Kusumoto, S., Kikuno, T., Matsumuto, K., Torii, K., 1997. A new fault localizing method for the program debugging process. Information and Software Technology 39, 271–284.

Lieberman, H., Fry, C., 1998. Software Visualization. MIT Press, Cap. ZStep95: A Reversible, Animated Source Code Stepper, pp. 277–292.

Linnenkugel, U., Müllerburg, M., Abril 1990. Test data selection for (software) integration testing. In: First International Conference on Systems Integration. Morristown, New Jersey.

Luckey, F. J., 1980. Understanding and debugging programs. International Journal of Man-Machines Studies , 189–202.

Lyle, J. R., Weiser, M., 1987. Automatic program bug location by program slicing. In: Proceedings of the 2nd International Conference on Computers and Applications.

Maldonado, J. C., 1991. Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. Tese de Doutorado — FEEC/Unicamp, Campinas.

Maldonado, J. C., Chaim, M. L., Jino, M., 1992a. Potential uses testing criteria: a step towards bridging the gap in the presence of infeasible paths. In: Proceedings of the XII International Conference of the Chilean Computer Society. Chilean Computer Society, Santiago, Chile.

Maldonado, J. C., Chaim, M. L., Jino, M., 1992b. Using the essential branch concept to support data-flow criteria application. In: Proceedings of the V International Conference on Software Engineering and its Applications. Toulouse, France.

Maldonado, J. C., Fabbri, S. C. P. F., 2001. Verificação e Validação de Software. In: Rocha, A. R. C., Maldonado, J. C., Weber, K. C. (Eds.), Qualidade de Software: Teoria e Prática. Prentice-Hall, São Paulo, SP, Brasil, pp. 66–73.

Maldonado, J. C., Virgílio, S. R., Chaim, M. L., Jino, M., 1992c. Critérios potenciais usos: Análise da aplicação de um benchmark. In: Anais do VI Simpósio Brasileiro de Engenharia de Software. Gramado, RS.

Marré, M., Bertolino, A., 1996. Unconstrained duas and their use in achieving all-uses coverage. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM Press, New York, U.S.A.

Marx, D. I. S., Frankl, F. G., 1999. Path-sensitive alias analysis for data flow testing. Software Testing, Verification and Reliability 9, 51–73.

Marx, D. I. S., Frankl, P. G., 1996. The path-wise approach to data flow testing with pointer variables. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM Press, New York, U.S.A.

Mathur, A. P., Wong, W. E., 1993. Comparing the fault detection effectiveness of mutation and data flow testing: An empirical study. Relatório Técnico SERC-TR-146-T, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, U.S.A.

Myers, G. J., 1979. The Art of Software Testing. Wiley-Inter-Science, New York.

Nishimatsu, A., Jihira, M., Kusumoto, S., Inoue, K., 1999. Call-Mark Slicing: A efficient and economical way of reducing slice. In: Proceedings of the 21st International Conference on Software Engineering. IEEE Computer Society Press, Los Angeles, CA, U.S.A.

Ntafos, S., 1984. A data flow oriented program testing strategy. IEEE Transactions on Software Engineering SE-10 (6), 795–803.

Offutt, A. J., Abdurazik, A., Alexander, R. T., September 2000. An analysis toof for coupling-based integration testing. In: IEEE Sixth International Conference on Engineering Complex Computer Systems. Tokyo, Japan.

Osterweil, L., Harrold, M. J., Clarke, L., 1996. Strategic directions in software quality. ACM Computing Surveys 28 (4), 738–750.

Ostrand, T. J., Balcer, M. J., 1988. The category-partition method for specifying and generating functional tests. Communications of the ACM 31 (6).

Ostrand, T. J., Weyuker, E. J., 1991. Data flow-based test adequacy analysis for languages with pointers. In: Proceedings of the Symposium on Software Testing, Analysis and Verification – TAV4. ACM Press, New York, U.S.A.

Pan, H., 1993. Software debugging with dynamic instrumentation and test-based knowledge. Tese de Doutorado, Purdue University.

Pan, H., Spafford, E. H., 1992. Toward automatic localization of software faults. In: Proceedings of the 10th Pacific Northwest Software Quality Conference. Portland, Oregon.

Pande, H., Landi, W., Ryder, B., 1994. Interprocedural def-use associations for C systems with single level pointers. IEEE Transactions on Software Engineering 20 (5), 789–810.

Pasquini, A., Crespo, A., Matrella, P., December 1996. Sensitivity of reliability-growth models to operational profiles erros vs testing accuracy. IEEE Transactions on Reliability R-45 (4), 531–540.

Pressman, R. S., 1994. Software Engineering: a practitioner's approach. McGraw-Hill, London.

Rapps, S., Weyuker, E. J., 1985. Selecting software test data using data flow information. IEEE Transactions on Software Engineering SE-14 (4), 367–375.

Rocha, A. R. C., 2001. Qualidade dos produtos de software. In: Rocha, A. R. C., Maldonado, J. C., Weber, K. C. (Eds.), Qualidade de Software: Teoria e Prática. Prentice-Hall, São Paulo, SP, Brasil, pp. 111–113.

Rosenblum, D. S., 1995. A practical approach to programming with assertions. IEEE Transactions on Software Engineering 21 (1), 19–31.

Shapiro, E. Y., 1983. Algorithmic Porgram Debugging. MIT Press, Cambridge, Massachussetts.

Shimomura, T., 1993. Critical slice-based fault localization for any type of error. IEICE Transactions on Information and Systems E76-D (6), 656–667.

Shimomura, T., Oki, Y., Chikaraishi, T., Ohta, T., 1995. An algorithmic fault-locating method for procedural languages and its implementation FIND. In: Second Workshop on Automated and Algorithmic Debugging. Saint-Malo, France.

Silva, J. B., 1995. PROTESTE+: Ambiente de Validação Automática da Qualidade de Software através de Técnicas de Teste e de Métricas de Complexidade. Dissertação de Mestrado — CPGCC-UFPRS, Porto Alegre.

Souza, S. R. S., 1996. Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de programas. Disserção de Mestrado, ICMC-USP, São Carlos, SP.

Souza, S. R. S., 2000. Validação de sistemas reativos: Definição e análise de critérios de teste. Tese de Doutorado, IFSC-USP, São Carlos, SP.

Staa, A. V., 2000. Programação Modular. Editora Campus, Rio de Janeiro, RJ.

Staa, A. V., 2001. Instrumentação. In: Rocha, A. R. C., Maldonado, J. C., Weber, K. C. (Eds.), Qualidade de Software: Teoria e Prática. Prentice-Hall, São Paulo, SP, Brasil, pp. 226–237.

Stallman, R. M., Pesch, R. H., 1999. Debugging with GDB: The GNU Source-Level Debugger, Seventh Edição. Free Software Foundation, Cambridge, MA.

Templer, K. S., Jefery, C. L., 1998. A configurable automatic instrumentation tool for ANSI C. In: Automate Software Engineering Conference.

Tip, F., 1995. A survey of program slicing techniques. Journal of Programming Languages 3, 121–189.

Tolmach, A., Appel, A. W., April 1995. A debugger for standard ML. Journal of Functional Programming 5 (2), 155–200.

Ural, H., Yang, B., 1988. A structural test selection criterion. Information Processing Letters 28, 157–163.

Vergilio, S. R., 1997. Critérios de teste de software restritos: Uma contribuição para a geração de dados de teste mais eficazes. Tese de Doutorado, FEEC/Unicamp, Campinas, SP, Brazil.

Vergilio, S. R., Maldonado, J. C., Jino, M., 1996. Um experimento de aplicação de critérios baseados em fluxo de dados de programas C. In: XV Congresso da Sociedade Brasileira de Computação e XXI Conferência Latinoamericana de Informática. Canela, RS.

Vessey, I., 1985. Expertise in debugging computer program: A process analysis. International Journal on Man-Machine Studies 23.

Vilela, P., 1998. Critérios Potenciais Usos de Integração: Definição e Análise. Tese de Doutorado — FEEC/Unicamp, Campinas, SP.

Vilela, P., Maldonado, J. C., Jino, M., 1997. Data flow based testing of programs with pointers: a strategy based on potential uses. In: Proceedings of the 10th International Software Quality Week — QW97. Software Research, Inc., San Francisco, California, U.S.A.

Viravan, C., 1994. Enhancing debugging technology. Tese de Doutorado, Purdue University.

Wah, K. S. H. T., 2000. A theoretical study of fault coupling. Software Testing, Verification and Reliability 10, 3–45.

Weiser, M., 1984. Program slicing. IEEE Transactions on Software Engineering SE-10 (4), 352–357.

Wong, W. E., 1993. On mutation and data flow. Tese de Doutorado, Purdue University, West Lafayette, Indiana, U.S.A.

Wong, W. E., Gokhale, S., Horgan, J. R., Trivedi, K. S., 1999. Locating program features using execution slices. In: ASSET — 1999 IEEE Symposium on Application-Specific Software Engineering and Technology. IEEE Computer Society Press, Richardson, Texas, U.S.A.

Apêndice A

Programa `Sort` do Sistema Unix

```
static  char *sccsid = "@(#)sort.c  4.13 (Berkeley) 11/12/87";
#include <sys/param.h>
#include <stdio.h>
#include <ctype.h>
#include <signal.h>
#include <sys/stat.h>

#define  L  2048
#define  N  7
#define  C  20
#ifndef pdp11
#define  MEM  (128*2048)
#else
#define  MEM  (16*2048)
#endif
#define NF  10

#define rline(mp)  (fgets((mp)->l, L, (mp)->b) == NULL)

FILE  *is, *os;
char  *dirtry[] = {"/usr/tmp", "/tmp", NULL};
char  **dirs;
char  file1[MAXPATHLEN];
char  *file = file1;
char  *filep;
int  nfiles;
unsigned  nlines;
unsigned  ntext;
int  *lspace;
char  *tspace;
int  cmp(), cmpa();
int  (*compare)() = cmpa;
char  *eol();
int  term();
int   mflg;
int  cflg;
int  uflg;
```

```
char  *outfil;
int unsafeout;  /*kludge to assure -m -o works*/
char  tabchar;
int   eargc;
char  **eargv;

char zero[256];

char  fold[256] = {
  0200,0201,0202,0203,0204,0205,0206,0207,
  0210,0211,0212,0213,0214,0215,0216,0217,
  0220,0221,0222,0223,0224,0225,0226,0227,
  0230,0231,0232,0233,0234,0235,0236,0237,
  0240,0241,0242,0243,0244,0245,0246,0247,
  0250,0251,0252,0253,0254,0255,0256,0257,
  0260,0261,0262,0263,0264,0265,0266,0267,
  0270,0271,0272,0273,0274,0275,0276,0277,
  0300,0301,0302,0303,0304,0305,0306,0307,
  0310,0311,0312,0313,0314,0315,0316,0317,
  0320,0321,0322,0323,0324,0325,0326,0327,
  0330,0331,0332,0333,0334,0335,0336,0337,
  0340,0341,0342,0343,0344,0345,0346,0347,
  0350,0351,0352,0353,0354,0355,0356,0357,
  0360,0361,0362,0363,0364,0365,0366,0367,
  0370,0371,0372,0373,0374,0375,0376,0377,
  0000,0001,0002,0003,0004,0005,0006,0007,
  0010,0011,0012,0013,0014,0015,0016,0017,
  0020,0021,0022,0023,0024,0025,0026,0027,
  0030,0031,0032,0033,0034,0035,0036,0037,
  0040,0041,0042,0043,0044,0045,0046,0047,
  0050,0051,0052,0053,0054,0055,0056,0057,
  0060,0061,0062,0063,0064,0065,0066,0067,
  0070,0071,0072,0073,0074,0075,0076,0077,
  0100,0101,0102,0103,0104,0105,0106,0107,
  0110,0111,0112,0113,0114,0115,0116,0117,
  0120,0121,0122,0123,0124,0125,0126,0127,
  0130,0131,0132,0133,0134,0135,0136,0137,
  0140,0101,0102,0103,0104,0105,0106,0107,
  0110,0111,0112,0113,0114,0115,0116,0117,
  0120,0121,0122,0123,0124,0125,0126,0127,
  0130,0131,0132,0173,0174,0175,0176,0177
};
char nofold[256] = {
  0200,0201,0202,0203,0204,0205,0206,0207,
  0210,0211,0212,0213,0214,0215,0216,0217,
  0220,0221,0222,0223,0224,0225,0226,0227,
  0230,0231,0232,0233,0234,0235,0236,0237,
  0240,0241,0242,0243,0244,0245,0246,0247,
  0250,0251,0252,0253,0254,0255,0256,0257,
  0260,0261,0262,0263,0264,0265,0266,0267,
  0270,0271,0272,0273,0274,0275,0276,0277,
  0300,0301,0302,0303,0304,0305,0306,0307,
```

```
  0310,0311,0312,0313,0314,0315,0316,0317,
  0320,0321,0322,0323,0324,0325,0326,0327,
  0330,0331,0332,0333,0334,0335,0336,0337,
  0340,0341,0342,0343,0344,0345,0346,0347,
  0350,0351,0352,0353,0354,0355,0356,0357,
  0360,0361,0362,0363,0364,0365,0366,0367,
  0370,0371,0372,0373,0374,0375,0376,0377,
  0000,0001,0002,0003,0004,0005,0006,0007,
  0010,0011,0012,0013,0014,0015,0016,0017,
  0020,0021,0022,0023,0024,0025,0026,0027,
  0030,0031,0032,0033,0034,0035,0036,0037,
  0040,0041,0042,0043,0044,0045,0046,0047,
  0050,0051,0052,0053,0054,0055,0056,0057,
  0060,0061,0062,0063,0064,0065,0066,0067,
  0070,0071,0072,0073,0074,0075,0076,0077,
  0100,0101,0102,0103,0104,0105,0106,0107,
  0110,0111,0112,0113,0114,0115,0116,0117,
  0120,0121,0122,0123,0124,0125,0126,0127,
  0130,0131,0132,0133,0134,0135,0136,0137,
  0140,0141,0142,0143,0144,0145,0146,0147,
  0150,0151,0152,0153,0154,0155,0156,0157,
  0160,0161,0162,0163,0164,0165,0166,0167,
  0170,0171,0172,0173,0174,0175,0176,0177
};

char  nonprint[256] = {
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
};

char  dict[256] = {
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
```

```
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,
  1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,
  1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1
};

struct  field {
  char *code;
  char *ignore;
  int nflg;
  int rflg;
  int bflg[2];
  int m[2];
  int n[2];
}  fields[NF];
struct field proto = {
  nofold+128,
  zero+128,
  0,
  1,
  0,0,
  0,-1,
  0,0
};
int  nfields;
int   error = 1;
char  *setfil();
char  *sbrk();
char  *brk();

#define  blank(c)  ((c) == ' ' || (c) == '\t')

main(argc, argv)
char **argv;
{
  register a;
  extern char end[1];
  char *ep;
  char *arg;
  struct field *p, *q;
  int i;

  copyproto();
  eargv = argv;
  while (--argc > 0) {
    if(***++argv == '-') for(arg = *argv;;) {
      switch(***++arg) {
```

```
    case '\0':
      if(arg[-1] == '-')
        eargv[eargc++] = "-";
      break;

    case 'o':
      if(--argc > 0)
        outfil = *++argv;
      continue;

    case 'T':
      if (--argc > 0)
        dirtry[0] = *++argv;
      continue;

    default:
      field(++*argv,nfields>0);
      break;
    }
    break;
  } else if (**argv == '+') {
    if(++nfields>=NF) {
      diag("too many keys","");
      exit(1);
    }
    copyproto();
    field(++*argv,0);
  } else
    eargv[eargc++] = *argv;
}
q = &fields[0];
for(a=1; a<=nfields; a++) {
  p = &fields[a];
  if(p->code != proto.code) continue;
  if(p->ignore != proto.ignore) continue;
  if(p->nflg != proto.nflg) continue;
  if(p->rflg != proto.rflg) continue;
  if(p->bflg[0] != proto.bflg[0]) continue;
  if(p->bflg[1] != proto.bflg[1]) continue;
  p->code = q->code;
  p->ignore = q->ignore;
  p->nflg = q->nflg;
  p->rflg = q->rflg;
  p->bflg[0] = p->bflg[1] = q->bflg[0];
}
if(eargc == 0)
  eargv[eargc++] = "-";
if(cflg && eargc>1) {
  diag("can check only 1 file","");
  exit(1);
}
safeoutfil();
```

```
  ep = end + MEM;
  lspace = (int *)sbrk(0);
  while((int)brk(ep) == -1)
    ep -= 512;
#ifndef  vax
  brk(ep -= 512);  /* for recursion */
#endif
  a = ep - (char*)lspace;
  nlines = (a-L);
  nlines /= (5*(sizeof(char *)/sizeof(char)));
  ntext = nlines * 4 * (sizeof(char *)/sizeof(char));
  tspace = (char *)(lspace + nlines);
  a = -1;
  for(dirs=dirtry; *dirs; dirs++) {
    sprintf(filep=file1, "%s/stm%05uaa", *dirs, getpid());
    while (*filep)
      filep++;
    filep -= 2;
    if ( (a=creat(file, 0600)) >=0)
      break;
  }
  if(a < 0) {
    diag("can't locate temp","");
    exit(1);
  }
  close(a);
  unlink(file);
  if (signal(SIGHUP, SIG_IGN) != SIG_IGN)
    signal(SIGHUP, term);
  if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, term);
  signal(SIGPIPE,term);
  if (signal(SIGTERM, SIG_IGN) != SIG_IGN)
    signal(SIGTERM,term);
  nfiles = eargc;
  if(!mflg && !cflg) {
    sort();
    fclose(stdin);
  }
  for(a = mflg|cflg?0:eargc; a+N<nfiles || unsafeout&&a<eargc; a=i) {
    i = a+N;
    if(i>=nfiles)
      i = nfiles;
    newfile();
    merge(a, i);
  }
  if(a != nfiles) {
    oldfile();
    merge(a, nfiles);
  }
  error = 0;
```

```
  term();
}

sort()
{
  register char *cp;
  register char **lp;
  register lines, text, len;
  int done = 0;
  int i = 0;
  char *f;
  char c;

  if((f = setfil(i++)) == NULL)
    is = stdin;
  else if((is = fopen(f, "r")) == NULL)
    cant(f);

  do {
    cp = tspace;
    lp = (char **)lspace;
    lines = nlines;
    text = ntext;
    while(lines > 0 && text > 0) {
      if(fgets(cp, L, is) == NULL) {
        if(i >= eargc) {
          ++done;
          break;
        }
        fclose(is);
        if((f = setfil(i++)) == NULL)
          is = stdin;
        else if((is = fopen(f, "r")) == NULL)
          cant(f);
        continue;
      }
      *lp++ = cp;
      len = strlen(cp) + 1; /* null terminate */
      if(cp[len - 2] != '\n')
        if (len == L) {
          diag("line too long (skipped): ", cp);
          while((c=getc(is)) != EOF && c != '\n')
            /* throw it away */;
          --lp;
          continue;
        } else {
          diag("missing newline before EOF in ",
            f ? f : "standard input");
          /* be friendly, append a newline */
          ++len;
          cp[len - 2] = '\n';
          cp[len - 1] = '\0';
```

```
        }
      cp += len;
      --lines;
      text -= len;
    }
    qsort((char **)lspace, lp);
    if(done == 0 || nfiles != eargc)
      newfile();
    else
      oldfile();
    clearerr(os);
    while(lp > (char **)lspace) {
      cp = *--lp;
      if(*cp)
        fputs(cp, os);
      if (ferror(os)) {
        error = 1;
        term();
      }
    }
    fclose(os);
  } while(done == 0);
}

struct merg
{
  char  l[L];
  FILE  *b;
} *ibuf[256];

merge(a,b)
{
  struct  merg  *p;
  register char  *cp, *dp;
  register  i;
  struct merg **ip, *jp;
  char  *f;
  int  j;
  int  k, l;
  int  muflg;

  p = (struct merg *)lspace;
  j = 0;
  for(i=a; i < b; i++) {
    f = setfil(i);
    if(f == 0)
      p->b = stdin;
    else if((p->b = fopen(f, "r")) == NULL)
      cant(f);
    ibuf[j] = p;
    if(!rline(p))  j++;
    p++;
```

```
}

do {
  i = j;
  qsort((char **)ibuf, (char **)(ibuf+i));
  l = 0;
  while(i--) {
    cp = ibuf[i]->l;
    if(*cp == '\0') {
      l = 1;
      if(rline(ibuf[i])) {
        k = i;
        while(++k < j)
          ibuf[k-1] = ibuf[k];
        j--;
      }
    }
  }
} while(l);

clearerr(os);
muflg = mflg & uflg | cflg;
i = j;
while(i > 0) {
  cp = ibuf[i-1]->l;
  if (!cflg && (uflg == 0 || muflg || i == 1 ||
    (*compare)(ibuf[i-1]->l,ibuf[i-2]->l))) {
    fputs(cp, os);
    if (ferror(os)) {
      error = 1;
      term();
    }
  }
  if(muflg){
    cp = ibuf[i-1]->l;
    dp = p->l;
    do {
    } while((*dp++ = *cp++) != '\n');
  }
  for(;;) {
    if(rline(ibuf[i-1])) {
      i--;
      if(i == 0)
        break;
      if(i == 1)
        muflg = uflg;
    }
    ip = &ibuf[i];
    while(--ip>ibuf&&(*compare)(ip[0]->l,ip[-1]->l)<0){
      jp = *ip;
      *ip = *(ip-1);
      *(ip-1) = jp;
```

```
      }
      if(!muflg)
        break;
      j = (*compare)(ibuf[i-1]->l,p->l);
      if(cflg) {
        if(j > 0)
          disorder("disorder:",ibuf[i-1]->l);
        else if(uflg && j==0)
          disorder("nonunique:",ibuf[i-1]->l);
      } else if(j == 0)
        continue;
      break;
    }
  }
  p = (struct merg *)lspace;
  for(i=a; i<b; i++) {
    fclose(p->b);
    p++;
    if(i >= eargc)
      unlink(setfil(i));
  }
  fclose(os);
}

disorder(s,t)
char *s, *t;
{
  register char *u;
  for(u=t; *u!='\n';u++) ;
  *u = 0;
  diag(s,t);
  term();
}

newfile()
{
  register char *f;

  f = setfil(nfiles);
  if((os=fopen(f, "w")) == NULL) {
    diag("can't create ",f);
    term();
  }
  nfiles++;
}

char *
setfil(i)
{

  if(i < eargc)
    if(eargv[i][0] == '-' && eargv[i][1] == '\0')
```

```
      return(0);
    else
      return(eargv[i]);
  i -= eargc;
  filep[0] = i/26 + 'a';
  filep[1] = i%26 + 'a';
  return(file);
}

oldfile()
{

  if(outfil) {
    if((os=fopen(outfil, "w")) == NULL) {
      diag("can't create ",outfil);
      term();
    }
  } else
    os = stdout;
}

safeoutfil()
{
  register int i;
  struct stat obuf,ibuf;

  if(!mflg||outfil==0)
    return;
  if(stat(outfil,&obuf)==-1)
    return;
  for(i=eargc-N;i<eargc;i++) {  /*-N is suff., not nec.*/
    if(stat(eargv[i],&ibuf)==-1)
      continue;
    if(obuf.st_dev==ibuf.st_dev&&
       obuf.st_ino==ibuf.st_ino)
      unsafeout++;
  }
}

cant(f)
char *f;
{

  perror(f);
  term();
}

diag(s,t)
char *s, *t;
{
  fputs("sort: ",stderr);
  fputs(s,stderr);
```

```
  fputs(t,stderr);
  fputs("\n",stderr);
}

term()
{
  register i;

  signal(SIGINT, SIG_IGN);
  signal(SIGHUP, SIG_IGN);
  signal(SIGTERM, SIG_IGN);
  if(nfiles == eargc)
    nfiles++;
  for(i=eargc; i<=nfiles; i++) {  /*<= in case of interrupt*/
    unlink(setfil(i));  /*with nfiles not updated*/
  }
  _exit(error);
}

cmp(i, j)
char *i, *j;
{
  register char *pa, *pb;
  char *skip();
  char *code, *ignore;
  int a, b;
  int k;
  char *la, *lb;
  register int sa;
  int sb;
  char *ipa, *ipb, *jpa, *jpb;
  struct field *fp;

  for(k = nfields>0; k<=nfields; k++) {
    fp = &fields[k];
    pa = i;
    pb = j;
    if(k) {
      la = skip(pa, fp, 1);
      pa = skip(pa, fp, 0);
      lb = skip(pb, fp, 1);
      pb = skip(pb, fp, 0);
    } else {
      la = eol(pa);
      lb = eol(pb);
    }
    if(fp->nflg) {
      if(tabchar) {
        if(pa<la&&*pa==tabchar)
          pa++;
        if(pb<lb&&*pb==tabchar)
          pb++;
```

```
      }
      while(blank(*pa))
        pa++;
      while(blank(*pb))
        pb++;
      sa = sb = fp->rflg;
      if(*pa == '-') {
        pa++;
        sa = -sa;
      }
      if(*pb == '-') {
        pb++;
        sb = -sb;
      }
      for(ipa = pa; ipa<la&&isdigit(*ipa); ipa++) ;
      for(ipb = pb; ipb<lb&&isdigit(*ipb); ipb++) ;
      jpa = ipa;
      jpb = ipb;
      a = 0;
      if(sa==sb)
        while(ipa > pa && ipb > pb)
          if(b = *--ipb - *--ipa)
            a = b;
      while(ipa > pa)
        if(*--ipa != '0')
          return(-sa);
      while(ipb > pb)
        if(*--ipb != '0')
          return(sb);
      if(a) return(a*sa);
      if(*(pa=jpa) == '.')
        pa++;
      if(*(pb=jpb) == '.')
        pb++;
      if(sa==sb)
        while(pa<la && isdigit(*pa)
           && pb<lb && isdigit(*pb))
          if(a = *pb++ - *pa++)
            return(a*sa);
      while(pa<la && isdigit(*pa))
        if(*pa++ != '0')
          return(-sa);
      while(pb<lb && isdigit(*pb))
        if(*pb++ != '0')
          return(sb);
      continue;
    }
    code = fp->code;
    ignore = fp->ignore;
loop:
    while(ignore[*pa])
      pa++;
```

```
    while(ignore[*pb])
      pb++;
    if(pa>=la || *pa=='\n')
      if(pb<lb && *pb!='\n')
        return(fp->rflg);
      else continue;
    if(pb>=lb || *pb=='\n')
      return(-fp->rflg);
    if((sa = code[*pb++]-code[*pa++]) == 0)
      goto loop;
    return(sa*fp->rflg);
  }
  if(uflg)
    return(0);
  return(cmpa(i, j));
}

cmpa(pa, pb)
register char *pa, *pb;
{
  while(*pa == *pb) {
    if(*pa++ == '\n')
      return(0);
    pb++;
  }
  return(
    *pa == '\n' ? fields[0].rflg:
    *pb == '\n' ?-fields[0].rflg:
    *pb > *pa   ? fields[0].rflg:
    -fields[0].rflg
  );
}

char *
skip(pp, fp, j)
struct field *fp;
char *pp;
{
  register i;
  register char *p;

  p = pp;
  if( (i=fp->m[j]) < 0)
    return(eol(p));
  while(i-- > 0) {
    if(tabchar != 0) {
      while(*p != tabchar)
        if(*p != '\n')
          p++;
        else goto ret;
      if(i>0||j==0)
        p++;
```

```
    } else {
      while(blank(*p))
        p++;
      while(!blank(*p))
        if(*p != '\n')
          p++;
        else goto ret;
    }
  }
  if(tabchar==0||fp->bflg[j])
    while(blank(*p))
      p++;
  i = fp->n[j];
  while(i-- > 0) {
    if(*p != '\n')
      p++;
    else goto ret;
  }
ret:
  return(p);
}

char *
eol(p)
register char *p;
{
  while(*p != '\n') p++;
  return(p);
}

copyproto()
{
  register i;
  register int *p, *q;

  p = (int *)&proto;
  q = (int *)&fields[nfields];
  for(i=0; i<sizeof(proto)/sizeof(*p); i++)
    *q++ = *p++;
}

field(s,k)
char *s;
{
  register struct field *p;
  register d;
  p = &fields[nfields];
  d = 0;
  for(; *s!=0; s++) {
    switch(*s) {
    case '\0':
      return;
```

```
    case 'b':
      p->bflg[k]++;
      break;

    case 'd':
      p->ignore = dict+128;
      break;

    case 'f':
      p->code = fold+128;
      break;
    case 'i':
      p->ignore = nonprint+128;
      break;

    case 'c':
      cflg = 1;
      continue;

    case 'm':
      mflg = 1;
      continue;

    case 'n':
      p->nflg++;
      break;
    case 't':
      tabchar = *++s;
      if(tabchar == 0) s--;
      continue;

    case 'r':
      p->rflg = -1;
      continue;
    case 'u':
      uflg = 1;
      break;

    case '.':
      if(p->m[k] == -1)  /* -m.n with m missing */
        p->m[k] = 0;
      d = &fields[0].n[0]-&fields[0].m[0];

    default:
      p->m[k+d] = number(&s);
    }
    compare = cmp;
  }
}

number(ppa)
```

```
char **ppa;
{
  int n;
  register char *pa;
  pa = *ppa;
  n = 0;
  while(isdigit(*pa)) {
    n = n*10 + *pa - '0';
    *ppa = pa++;
  }
  return(n);
}

#define qsexc(p,q) t= *p;*p= *q;*q=t
#define qstexc(p,q,r) t= *p;*p= *r;*r= *q;*q=t

qsort(a,l)
char **a, **l;
{
  register char **i, **j;
  char **k;
  char **lp, **hp;
  int c;
  char *t;
  unsigned n;



start:
  if((n=l-a) <= 1)
    return;


  n /= 2;
  hp = lp = a+n;
  i = a;
  j = l-1;


  for(;;) {
    if(i < lp) {
      if((c = (*compare)(*i, *lp)) == 0) {
        --lp;
        qsexc(i, lp);
        continue;
      }
      if(c < 0) {
        ++i;
        continue;
      }
    }
```

```
loop:
    if(j > hp) {
      if((c = (*compare)(*hp, *j)) == 0) {
        ++hp;
        qsexc(hp, j);
        goto loop;
      }
      if(c > 0) {
        if(i == lp) {
          ++hp;
          qstexc(i, hp, j);
          i = ++lp;
          goto loop;
        }
        qsexc(i, j);
        --j;
        ++i;
        continue;
      }
      --j;
      goto loop;
    }


    if(i == lp) {
      if(uflg)
        for(k=lp+1; k<=hp;) **k++ = '\0';
      if(lp-a >= l-hp) {
        qsort(hp+1, l);
        l = lp;
      } else {
        qsort(a, lp);
        a = hp+1;
      }
      goto start;
    }


    --lp;
    qstexc(j, lp, i);
    j = --hp;
  }
}
```

Apêndice B

Dados Completos sobre o Uso de Requisitos de Teste na Depuração

As Tabelas B.1, B.2 e B.3 contêm os resultados das heurísticas utilizando os requisitos de teste dos critérios todos ramos, todos usos e todos potenciais usos considerando o cenário TP-C. As Tabelas B.4, B.5 e B.6 analogamente contêm os resultados para o cenário GT-C; e as Tabelas B.7, B.8 e B.9 os resultados para o cenário MT-C. A estrutura das tabelas é exemplificada a seguir.

Considere-se os resultados da aplicação do par (H2, todos ramos) para o defeito 4 descritos na Tabela B.1. A medida de inclusão $Inc_{\text{sítio}}$ obtida foi de 20%, o que significa que em apenas 20% dos conjuntos de casos teste nos quais o par (H2, todos ramos) foi aplicado o sítio do defeito foi atingido. Para estes casos em que (H2, todos ramos) foi bem sucedido, $Cmd_{\text{médio}}$ e Ef_{cmd} obtidos foram, respectivamente, 72,17 e 48,94%, significando que, em média, 72,17 comandos foram identificados, sendo que eles representam 48,94% do tamanho dos conjuntos de comandos selecionados por H1.

Com relação às medidas baseadas em *rt-re*s, a medida de inclusão Inc_{erro} obtida foi de 100%, ou seja, todos os conjuntos $CE_{(H,C,T_i)}$ incluem pelo menos um *rt-re*. Em termos de eficiência, $CE_{\text{médio}}$ e Ef_{erro} obtidos foram, respectivamente, 23,07 e 55,97%, o que significa que o número médio de ramos selecionados para investigação foi 23,07 ramos sendo que 55,97% deles eram reveladores de erros.

Tabela B.1: Resultados da aplicação do par (heurística, todos ramos) no cenário TP-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Ramos	H1	$Inc_{\text{sítio}}$	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	180,30	154,90	139,27	132,60	157,60	143,97	143,23	123,53	154,20	159,93	138,60
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	75,93	68,10	57,53	54,30	64,83	59,13	60,60	55,57	65,57	63,37	56,53
		Ef_{erro}	29,93	4,55	23,72	37,87	51,84	11,26	17,05	8,34	6,57	27,85	21,97
	H2	$Inc_{\text{sítio}}$	100,00	100,00	86,67	20,00	0,00	96,67	100,00	60,00	100,00	73,33	96,67
		$Cmd_{\text{médio}}$	95,33	77,07	63,23	72,17	0,00	67,52	66,27	53,06	71,57	84,27	59,34
		Ef_{cmd}	52,66	48,87	43,97	48,94	0,00	46,16	44,99	43,25	45,72	52,52	42,96
		Inc_{erro}	100,00	100,00	96,67	100,00	100,00	96,67	100,00	76,67	100,00	90,00	96,67
		$CE_{\text{médio}}$	41,53	35,30	24,34	23,07	29,27	27,93	28,43	24,09	30,63	29,78	24,41
		Ef_{erro}	40,15	9,60	44,23	55,97	62,87	21,55	23,36	10,22	11,10	36,78	29,39
	H3	$Inc_{\text{sítio}}$	86,67	26,67	63,33	3,33	0,00	96,67	90,00	13,33	100,00	70,00	100,00
		$Cmd_{\text{médio}}$	24,00	11,75	30,58	57,00	0,00	18,97	32,52	48,25	41,30	43,48	12,30
		Ef_{cmd}	13,26	7,63	22,50	61,96	0,00	12,89	21,70	44,65	26,67	27,80	8,87
		Inc_{erro}	86,67	26,67	73,33	96,67	43,33	100,00	90,00	30,00	100,00	76,67	100,00
		$CE_{\text{médio}}$	10,27	5,50	12,32	5,90	18,85	7,40	13,93	20,78	18,70	16,30	4,40
		Ef_{erro}	43,83	50,12	60,87	82,42	67,96	64,52	24,86	11,31	19,51	35,72	42,14
	H4	$Inc_{\text{sítio}}$	96,67	90,00	70,00	3,33	10,00	96,67	93,33	30,00	100,00	76,67	100,00
		$Cmd_{\text{médio}}$	34,93	11,52	48,81	94,00	175,33	45,03	56,39	88,44	76,57	82,78	18,23
		Ef_{cmd}	19,25	7,82	36,81	102,17	103,72	30,78	37,51	70,90	49,25	52,06	13,19
		Inc_{erro}	96,67	90,00	86,67	96,67	50,00	100,00	93,33	50,00	100,00	90,00	100,00
		$CE_{\text{médio}}$	15,83	5,70	19,27	10,93	29,27	15,73	23,29	29,60	31,93	28,22	7,13
		Ef_{erro}	35,02	37,25	49,93	83,21	67,95	36,38	20,64	11,64	12,73	38,64	24,28

Tabela B.2: Resultados da aplicação do par (heurística, todos usos) no cenário TP-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Usos	H1	$Inc_{\text{sítio}}$	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	283,13	223,30	234,63	216,53	248,73	233,87	232,17	213,80	243,90	263,83	227,33
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	831,60	512,27	540,57	534,63	624,70	537,63	546,47	437,47	608,03	692,50	524,30
		Ef_{erro}	25,80	2,94	26,92	35,46	50,37	9,99	20,22	15,74	6,89	33,81	25,21
	H2	$Inc_{\text{sítio}}$	100,00	100,00	83,33	80,00	43,33	100,00	100,00	73,33	100,00	93,33	100,00
		$Cmd_{\text{médio}}$	214,47	142,30	141,80	111,58	149,62	141,90	145,10	129,41	164,50	170,71	148,57
		Ef_{cmd}	75,68	63,91	59,90	51,70	58,97	60,05	61,95	59,26	66,44	64,76	65,35
		Inc_{erro}	100,00	100,00	90,00	100,00	100,00	100,00	100,00	93,33	100,00	100,00	100,00
		$CE_{\text{médio}}$	531,77	286,70	290,96	224,33	324,73	280,53	301,50	185,93	387,40	423,07	277,87
		Ef_{erro}	30,55	5,59	30,66	41,46	59,13	15,47	27,58	8,40	8,72	38,94	23,83
	H3	$Inc_{\text{sítio}}$	56,67	50,00	40,00	10,00	3,33	93,33	100,00	33,33	100,00	73,33	100,00
		$Cmd_{\text{médio}}$	98,35	51,73	80,75	128,00	173,00	66,86	74,47	120,90	91,27	104,91	44,27
		Ef_{cmd}	35,49	22,60	33,92	59,41	64,79	27,68	31,73	53,89	36,22	39,97	19,49
		Inc_{erro}	83,33	90,00	63,33	100,00	76,67	100,00	100,00	76,67	100,00	86,67	100,00
		$CE_{\text{médio}}$	123,52	43,04	128,89	67,20	109,09	77,90	133,60	124,78	167,13	236,58	84,80
		Ef_{erro}	32,72	53,59	39,17	65,06	71,24	39,40	23,53	11,72	12,47	35,67	42,74
	H4	$Inc_{\text{sítio}}$	86,67	86,67	56,67	13,33	6,67	93,33	100,00	73,33	100,00	83,33	100,00
		$Cmd_{\text{médio}}$	105,08	49,96	111,41	151,00	194,50	92,39	96,87	121,68	118,43	132,40	55,13
		Ef_{cmd}	37,58	21,93	47,80	69,64	82,25	38,59	41,52	55,86	47,41	50,17	24,27
		Inc_{erro}	93,33	100,00	73,33	96,67	93,33	100,00	100,00	90,00	100,00	96,67	100,00
		$CE_{\text{médio}}$	190,75	68,53	224,18	107,90	161,86	161,97	212,57	226,81	284,70	325,72	119,23
		Ef_{erro}	34,02	41,39	34,79	57,57	67,60	29,69	24,17	9,33	10,59	38,24	30,89

Tabela B.3: Resultados da aplicação do par (heurística, todos potenciais usos) no cenário TP-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Potenciais Usos	H1	$Inc_{\text{sítio}}$	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	279,73	223,37	238,80	220,03	268,80	228,80	241,90	210,77	244,13	251,50	223,37
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	984,10	619,93	577,27	494,53	800,03	586,67	719,77	536,13	755,33	683,80	600,30
		Ef_{erro}	33,95	2,86	16,16	29,40	47,15	11,42	13,31	6,86	7,92	20,31	12,41
	H2	$Inc_{\text{sítio}}$	100,00	100,00	93,33	86,67	100,00	100,00	100,00	100,00	100,00	90,00	100,00
		$Cmd_{\text{médio}}$	219,00	164,77	171,54	145,81	200,77	161,33	172,00	141,60	182,00	187,44	169,80
		Ef_{cmd}	78,13	73,46	71,72	66,48	74,73	69,85	70,72	67,14	74,22	74,40	76,17
		Inc_{erro}	100,00	100,00	96,67	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	680,63	363,50	324,00	244,40	472,83	289,40	442,13	266,53	491,97	390,80	351,13
		Ef_{erro}	43,53	5,45	23,35	43,29	54,10	16,89	17,43	8,97	11,97	25,89	14,40
	H3	$Inc_{\text{sítio}}$	93,33	56,67	46,67	0,00	23,33	96,67	96,67	86,67	100,00	76,67	100,00
		$Cmd_{\text{médio}}$	114,50	57,94	97,79	0,00	121,00	99,21	119,17	79,96	118,47	152,00	59,03
		Ef_{cmd}	41,04	25,26	40,50	0,00	42,57	43,73	49,25	38,16	48,37	60,05	26,58
		Inc_{erro}	96,67	96,67	66,67	100,00	66,67	100,00	96,67	86,67	100,00	93,33	100,00
		$CE_{\text{médio}}$	210,17	50,62	126,80	48,10	113,15	130,57	253,62	110,15	300,97	268,54	72,40
		Ef_{erro}	44,50	51,55	41,22	76,88	60,82	40,09	19,60	11,40	15,02	24,41	42,44
	H4	$Inc_{\text{sítio}}$	100,00	80,00	46,67	6,67	23,33	96,67	100,00	96,67	100,00	90,00	100,00
		$Cmd_{\text{médio}}$	123,80	70,83	132,36	101,50	138,71	128,79	149,73	99,03	140,33	178,11	71,33
		Ef_{cmd}	44,27	30,78	54,74	48,04	49,27	56,15	61,84	47,35	57,31	70,07	32,12
		Inc_{erro}	100,00	100,00	73,33	100,00	76,67	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	280,67	92,73	220,77	69,37	173,57	258,07	387,13	189,03	431,47	398,53	102,50
		Ef_{erro}	47,34	38,09	33,70	74,90	54,95	28,62	16,77	11,68	12,27	27,24	31,18

Tabela B.4: Resultados da aplicação do par (heurística, todos ramos) no cenário GT-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Ramos	H1	$Inc_{\text{sítio}}$	100,00	100,00	56,67	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	151,93	127,77	89,41	118,37	134,77	70,13	46,77	60,43	46,80	92,20	128,87
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	63,77	57,73	28,60	48,00	54,67	27,97	21,23	27,43	21,60	34,40	51,97
		Ef_{erro}	33,73	4,97	19,95	36,22	57,00	11,68	4,85	13,73	5,91	32,38	21,49
	H2	$Inc_{\text{sítio}}$	100,00	100,00	100,00	3,33	0,00	86,67	100,00	96,67	100,00	86,67	100,00
		$Cmd_{\text{médio}}$	87,83	78,50	78,10	53,00	0,00	41,58	75,60	57,90	87,53	97,85	47,20
		Ef_{cmd}	61,97	67,21	155,65	85,48	0,00	64,07	167,14	112,24	195,12	117,98	37,22
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	86,67	100,00	100,00	100,00	93,33	100,00
		$CE_{\text{médio}}$	41,33	35,93	35,77	20,97	27,43	19,62	34,80	27,30	40,63	37,57	20,23
		Ef_{erro}	46,33	9,77	27,42	47,78	49,03	14,02	17,46	10,68	9,57	33,59	17,95
	H3	$Inc_{\text{sítio}}$	56,67	50,00	43,33	0,00	0,00	80,00	100,00	40,00	100,00	86,67	100,00
		$Cmd_{\text{médio}}$	7,06	2,13	2,08	0,00	0,00	1,58	6,27	1,83	6,67	9,12	5,07
		Ef_{cmd}	5,08	2,49	4,52	0,00	0,00	2,67	13,30	4,51	13,96	11,63	3,91
		Inc_{erro}	56,67	50,00	73,33	96,67	53,33	100,00	100,00	43,33	100,00	96,67	100,00
		$CE_{\text{médio}}$	4,41	1,07	1,68	2,38	2,44	1,23	2,67	2,31	3,13	3,31	2,03
		Ef_{erro}	78,98	96,67	95,45	95,98	94,79	100,00	54,72	87,18	55,90	62,47	66,39
	H4	$Inc_{\text{sítio}}$	96,67	100,00	50,00	0,00	0,00	80,00	100,00	60,00	100,00	96,67	100,00
		$Cmd_{\text{médio}}$	11,86	6,27	4,80	0,00	0,00	10,42	11,23	6,78	11,37	16,83	8,70
		Ef_{cmd}	7,81	6,12	9,75	0,00	0,00	18,49	24,14	15,39	24,52	20,54	6,91
		Inc_{erro}	96,67	100,00	80,00	96,67	60,00	100,00	100,00	63,33	100,00	100,00	100,00
		$CE_{\text{médio}}$	6,66	3,23	3,75	4,83	4,78	3,00	4,87	4,26	5,30	6,60	3,60
		Ef_{erro}	60,30	49,04	75,14	96,46	89,75	53,11	23,28	48,07	27,10	51,06	28,81

Tabela B.5: Resultados da aplicação do par (heurística, todos usos) no cenário GT-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Usos	H1	$Inc_{\text{sítio}}$	100,00	100,00	40,00	100,00	100,00	6,67	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	202,30	144,17	142,92	134,03	164,10	111,50	110,87	128,73	109,37	140,50	208,70
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	510,73	216,73	197,73	251,47	310,50	148,53	134,17	209,77	123,93	249,43	474,50
		Ef_{erro}	23,85	3,05	36,10	28,07	53,16	5,73	4,27	27,66	1,27	27,90	25,22
	H2	$Inc_{\text{sítio}}$	100,00	100,00	100,00	73,33	53,33	100,00	100,00	100,00	100,00	70,00	100,00
		$Cmd_{\text{médio}}$	190,47	160,03	143,20	118,64	162,38	133,87	159,03	165,17	163,57	180,19	128,90
		Ef_{cmd}	94,81	113,52	120,49	94,49	107,91	117,79	144,31	128,55	150,10	128,74	62,35
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	96,67	100,00
		$CE_{\text{médio}}$	494,73	380,73	314,73	260,83	299,20	225,63	388,80	346,93	403,37	378,59	236,53
		Ef_{erro}	37,89	5,88	19,15	28,06	44,43	11,94	16,79	7,57	5,78	28,26	19,40
	H3	$Inc_{\text{sítio}}$	46,67	10,00	40,00	0,00	0,00	0,00	100,00	66,67	100,00	90,00	100,00
		$Cmd_{\text{médio}}$	27,00	7,00	7,75	0,00	0,00	0,00	14,17	10,25	8,93	10,93	16,13
		Ef_{cmd}	14,06	5,39	6,36	0,00	0,00	0,00	12,87	8,03	8,06	8,13	7,86
		Inc_{erro}	73,33	96,67	86,67	100,00	90,00	0,00	100,00	76,67	100,00	100,00	100,00
		$CE_{\text{médio}}$	19,05	2,62	4,19	7,93	2,48	0,00	13,97	9,43	5,90	6,67	27,07
		Ef_{erro}	72,93	91,19	86,42	92,53	100,00	0,00	31,64	82,14	40,37	57,33	84,98
	H4	$Inc_{\text{sítio}}$	76,67	13,33	40,00	0,00	0,00	26,67	100,00	83,33	100,00	93,33	100,00
		$Cmd_{\text{médio}}$	38,74	13,00	11,83	0,00	0,00	15,12	20,33	23,80	32,80	20,04	19,27
		Ef_{cmd}	19,55	10,44	9,73	0,00	0,00	13,73	18,36	18,57	29,42	14,73	9,38
		Inc_{erro}	100,00	96,67	96,67	100,00	100,00	46,67	100,00	96,67	100,00	100,00	100,00
		$CE_{\text{médio}}$	39,07	5,52	9,41	15,80	6,23	6,07	22,83	23,72	96,70	20,20	59,20
		Ef_{erro}	52,49	87,79	69,87	91,16	91,07	42,55	18,82	43,92	12,71	56,65	38,86

Tabela B.6: Resultados da aplicação do par (heurística, todos potenciais) no cenário GT-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Potenciais Usos	H1	$Inc_{\text{sítio}}$	100,00	100,00	10,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	210,87	121,87	106,33	114,10	139,37	108,13	99,27	121,50	94,97	105,73	197,63
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	557,20	229,80	128,27	185,17	258,07	134,03	173,37	187,17	139,70	144,47	491,07
		Ef_{erro}	35,77	4,49	6,55	25,29	35,10	3,35	7,94	10,32	3,56	9,45	12,95
	H2	$Inc_{\text{sítio}}$	100,00	100,00	100,00	93,33	100,00	100,00	100,00	100,00	100,00	60,00	100,00
		$Cmd_{\text{médio}}$	215,53	194,60	151,20	154,04	185,33	154,10	192,43	187,00	196,43	215,78	158,07
		Ef_{cmd}	103,42	161,51	170,24	147,41	136,28	142,92	194,49	154,28	207,34	210,17	80,97
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	90,00	100,00
		$CE_{\text{médio}}$	706,50	496,27	394,73	293,80	416,67	260,60	550,93	467,17	597,00	404,48	302,57
		Ef_{erro}	43,16	6,02	14,45	31,39	41,40	17,70	13,08	9,48	5,78	20,28	15,93
	H3	$Inc_{\text{sítio}}$	93,33	20,00	33,33	0,00	0,00	100,00	100,00	56,67	100,00	96,67	100,00
		$Cmd_{\text{médio}}$	8,71	8,00	8,00	0,00	0,00	3,53	22,20	6,76	10,73	11,79	22,23
		Ef_{cmd}	4,18	7,18	9,08	0,00	0,00	3,17	22,43	5,58	11,36	11,59	11,35
		Inc_{erro}	96,67	100,00	80,00	100,00	90,00	100,00	100,00	56,67	100,00	100,00	100,00
		$CE_{\text{médio}}$	2,90	3,53	2,83	10,87	1,89	2,43	29,37	2,94	7,67	6,50	21,63
		Ef_{erro}	94,66	100,00	97,50	97,80	100,00	97,79	48,28	81,51	86,35	56,35	93,07
	H4	$Inc_{\text{sítio}}$	96,67	43,33	36,67	0,00	0,00	100,00	100,00	96,67	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	30,45	17,85	13,64	0,00	0,00	9,17	24,30	11,48	17,47	18,73	20,37
		Ef_{cmd}	14,97	15,20	15,29	0,00	0,00	8,25	24,55	9,51	18,41	18,15	10,51
		Inc_{erro}	96,67	100,00	86,67	100,00	90,00	100,00	100,00	96,67	100,00	100,00	100,00
		$CE_{\text{médio}}$	34,31	7,47	6,04	18,20	7,56	6,03	40,80	5,83	21,37	11,60	30,83
		Ef_{erro}	68,11	88,89	66,94	99,05	97,16	77,41	34,97	70,97	33,02	48,37	71,27

Tabela B.7: Resultados da aplicação do par (heurística, todos ramos) no cenário MT-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Ramos	H1	$Inc_{\text{sítio}}$	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	175,47	140,80	132,63	126,23	158,70	142,47	145,07	108,50	158,27	151,13	137,87
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	74,70	61,87	55,63	51,67	64,40	58,10	61,47	49,77	73,33	61,33	56,80
		Ef_{erro}	30,40	4,31	20,78	37,20	52,40	11,07	15,72	9,63	8,35	25,02	22,33
	H2	$Inc_{\text{sítio}}$	100,00	100,00	86,67	10,00	0,00	86,67	100,00	100,00	100,00	60,00	100,00
		$Cmd_{\text{médio}}$	79,13	62,53	54,85	57,33	0,00	43,96	79,23	49,47	74,57	77,28	49,33
		Ef_{cmd}	45,08	45,52	42,07	52,42	0,00	28,82	53,97	45,27	46,64	50,34	35,57
		Inc_{erro}	100,00	100,00	96,67	100,00	96,67	86,67	100,00	100,00	100,00	70,00	100,00
		$CE_{\text{médio}}$	37,43	29,57	24,10	20,77	18,55	20,62	34,53	23,83	34,47	28,62	21,93
		Ef_{erro}	45,26	7,92	26,33	52,73	67,99	14,64	19,20	10,69	11,56	32,13	20,70
	H3	$Inc_{\text{sítio}}$	43,33	66,67	70,00	0,00	0,00	26,67	80,00	0,00	76,67	43,33	100,00
		$Cmd_{\text{médio}}$	8,54	3,45	3,24	0,00	0,00	10,50	5,04	0,00	4,61	7,00	2,20
		Ef_{cmd}	5,10	2,48	2,33	0,00	0,00	6,40	3,44	0,00	3,06	4,58	1,62
		Inc_{erro}	43,33	66,67	73,33	100,00	46,67	96,67	80,00	3,33	76,67	43,33	100,00
		$CE_{\text{médio}}$	5,54	1,65	2,00	2,27	2,36	2,00	2,79	3,00	3,35	3,46	1,07
		Ef_{erro}	60,69	72,92	79,17	96,67	82,74	82,18	58,91	33,33	55,11	36,15	97,78
	H4	$Inc_{\text{sítio}}$	90,00	93,33	76,67	0,00	0,00	26,67	93,33	0,00	90,00	63,33	100,00
		$Cmd_{\text{médio}}$	14,67	8,21	6,61	0,00	0,00	18,88	10,64	0,00	9,70	14,53	8,03
		Ef_{cmd}	8,50	5,78	4,88	0,00	0,00	12,78	7,40	0,00	6,37	9,36	5,88
		Inc_{erro}	90,00	93,33	80,00	100,00	53,33	96,67	93,33	3,33	90,00	63,33	100,00
		$CE_{\text{médio}}$	8,41	3,86	3,83	4,50	4,94	3,86	5,75	5,00	6,00	7,21	3,03
		Ef_{erro}	42,22	41,54	71,19	89,83	76,25	45,59	29,46	20,00	33,99	24,14	33,06

Tabela B.8: Resultados da aplicação do par (heurística, todos usos) no cenário MT-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Usos	H1	$Inc_{\text{sítio}}$	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	276,10	230,93	241,67	214,60	253,63	231,93	233,83	215,53	242,20	249,87	231,43
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	818,97	550,03	551,83	517,37	646,70	508,87	559,97	436,93	607,37	616,97	544,37
		Ef_{erro}	23,89	3,08	33,28	33,14	51,65	9,93	21,40	15,16	7,86	28,19	22,85
	H2	$Inc_{\text{sítio}}$	100,00	100,00	93,33	90,00	66,67	100,00	100,00	96,67	100,00	66,67	100,00
		$Cmd_{\text{médio}}$	188,20	139,53	115,54	117,15	134,75	115,77	133,93	130,00	137,80	161,55	145,37
		Ef_{cmd}	68,61	61,07	48,01	55,59	52,67	49,13	57,90	60,07	57,84	64,62	62,66
		Inc_{erro}	100,00	100,00	93,33	100,00	100,00	100,00	100,00	96,67	100,00	96,67	100,00
		$CE_{\text{médio}}$	479,43	258,97	164,29	210,97	203,57	148,10	267,73	242,55	266,67	303,28	260,77
		Ef_{erro}	35,40	5,97	17,04	34,92	57,71	8,56	19,05	8,07	11,90	28,00	18,00
	H3	$Inc_{\text{sítio}}$	16,67	3,33	43,33	0,00	0,00	23,33	96,67	0,00	76,67	26,67	100,00
		$Cmd_{\text{médio}}$	26,80	9,00	11,62	0,00	0,00	20,43	13,03	0,00	16,00	17,00	16,00
		Ef_{cmd}	9,99	4,29	4,74	0,00	0,00	8,87	5,82	0,00	6,61	6,58	6,93
		Inc_{erro}	96,67	73,33	63,33	73,33	26,67	70,00	56,67	0,00	76,67	30,00	100,00
		$CE_{\text{médio}}$	16,12	3,59	4,53	3,09	2,50	5,00	12,65	0,00	9,35	8,22	27,00
		Ef_{erro}	54,39	81,10	90,01	79,94	93,75	96,66	34,65	0,00	52,32	28,62	85,19
	H4	$Inc_{\text{sítio}}$	53,33	53,33	46,67	0,00	0,00	40,00	96,67	0,00	100,00	33,33	100,00
		$Cmd_{\text{médio}}$	45,25	18,44	17,00	0,00	0,00	21,25	21,07	0,00	27,83	26,50	19,00
		Ef_{cmd}	16,52	8,00	6,93	0,00	0,00	9,33	9,28	0,00	11,46	10,37	8,22
		Inc_{erro}	83,33	96,67	66,67	86,67	43,33	86,67	96,67	0,00	100,00	43,33	100,00
		$CE_{\text{médio}}$	51,32	10,76	12,75	11,96	6,85	6,77	20,59	0,00	26,53	17,38	59,00
		Ef_{erro}	38,73	60,72	72,83	76,40	75,20	64,35	32,25	0,00	36,20	18,55	38,98

Tabela B.9: Resultados da aplicação do par (heurística, todos potenciais usos) no cenário MT-C

Requisitos de Teste	Heurística		Defeitos										
			1	2	3	4	5	6	7	8	9	10	11
Todos Potenciais Usos	H1	$Inc_{\text{sítio}}$	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$Cmd_{\text{médio}}$	282,73	240,37	242,47	219,23	254,87	232,13	240,10	204,87	246,67	250,97	224,27
		Ef_{cmd}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	1027,13	702,40	618,40	529,67	720,17	611,80	697,50	510,90	745,47	614,93	584,20
		Ef_{erro}	34,64	2,80	16,10	25,10	40,17	11,42	13,21	6,88	6,75	20,86	14,69
	H2	$Inc_{\text{sítio}}$	100,00	100,00	100,00	96,67	96,67	100,00	100,00	100,00	100,00	76,67	100,00
		$Cmd_{\text{médio}}$	212,83	180,20	154,70	142,66	170,83	138,80	160,30	156,10	169,27	189,78	160,80
		Ef_{cmd}	75,27	74,91	64,06	65,11	67,36	59,25	67,01	76,16	69,40	75,05	71,56
		Inc_{erro}	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
		$CE_{\text{médio}}$	673,33	414,13	283,53	251,30	333,77	235,30	430,07	302,33	449,90	339,67	307,27
		Ef_{erro}	43,37	4,99	15,44	33,82	43,05	14,70	13,21	8,93	9,73	24,56	16,11
	H3	$Inc_{\text{sítio}}$	76,67	16,67	6,67	0,00	0,00	50,00	73,33	0,00	93,33	0,00	100,00
		$Cmd_{\text{médio}}$	14,04	32,80	8,50	0,00	0,00	25,87	18,27	0,00	17,46	0,00	20,27
		Ef_{cmd}	4,95	12,02	3,58	0,00	0,00	10,37	7,89	0,00	7,18	0,00	9,07
		Inc_{erro}	76,67	100,00	13,33	86,67	26,67	96,67	73,33	0,00	93,33	10,00	100,00
		$CE_{\text{médio}}$	9,91	4,30	2,75	4,31	5,50	11,72	22,32	0,00	21,46	2,00	22,43
		Ef_{erro}	61,10	75,73	90,00	80,47	100,00	86,40	44,28	0,00	48,78	83,33	90,41
	H4	$Inc_{\text{sítio}}$	96,67	16,67	10,00	0,00	0,00	50,00	100,00	30,00	100,00	3,33	100,00
		$Cmd_{\text{médio}}$	42,62	41,60	20,67	0,00	0,00	44,20	30,37	22,11	33,57	25,00	20,43
		Ef_{cmd}	15,03	15,22	7,63	0,00	0,00	17,88	12,90	10,34	13,73	9,62	9,15
		Inc_{erro}	96,67	100,00	20,00	90,00	33,33	96,67	100,00	30,00	100,00	30,00	100,00
		$CE_{\text{médio}}$	56,59	9,20	10,17	16,85	10,90	21,79	51,17	16,33	54,80	8,00	36,30
		Ef_{erro}	50,74	72,17	62,02	74,10	86,50	71,84	32,29	11,33	38,83	40,13	60,27

Apêndice C

Descrição do Depurador `Gdb`

A seguir é apresentada uma descrição simplificada das principais funções do depurador simbólico `gdb`. Maiores detalhes sobre este depurador é fornecido por Stallman e Pesch (1999).

C.1 Introdução

`Gdb` é o depurador simbólico do projeto GNU desenvolvido para apoiar a depuração de programas escritos em linguagem C ou C++. O objetivo principal de um depurador como `gdb` é permitir o rastreamento de *eventos* e a *inspeção* do estado do programa durante a execução. Para que o programador possa testar hipóteses com relação à localização ou à correção do defeito, `gdb` permite ainda a alteração do estado do programa.

Assim, para atingir os objetivos acima, o depurador `gdb` possui comandos que implementam as seguintes funções principais: *parada e execução do programa*; *observação e modificação do estado do programa*; e *definição de comandos pelo usuário*. A seguir são apresentados os principais comandos que realizam essas funções.

C.2 Parada e Execução do Programa

C.2.1 Parada

Uma das principais funções de um depurador simbólico é *parar* a execução de um programa quando ocorre um determinado evento. Este evento pode ser o alcance de uma determinada linha do programa, a modificação do valor de uma variável ou a ocorrência de um sinal.

Usando o comando `break`, o programador pode ajustar um *breakpoint* no programa. O *breakpoint* faz com que o programa pare sempre que um certo ponto do programa é atingido. Este comando permite ainda adicionar condições que refinam as situações em que o programa pára. Um *watchpoint* é um tipo especial de *breakpoint* que pára o programa quando o valor de uma expressão muda. Um

catchpoint é outro tipo especial de *breakpoint* que pára o programa quando um certo tipo de evento acontece como, por exemplo, a ocorrência de uma exceção ou o carregamento de uma biblioteca. Os *watchpoint*s e *catchpoint*s são ajustados, respectivamente, pelos comandos `watch` e catch.

C.2.2 Execução

Os eventos rastreados por `gdb` podem estar associados também à execução do programa. Exemplos de eventos desse tipo são o alcance da *próxima* linha do programa, do ponto de retorno de uma função etc. Para rastrear estes eventos, `gdb` fornece os comandos `step`, `next`, `finish` e `continue`. O comando `step` executa o programa até encontrar uma linha diferente quando então a execução é interrompida e o controle retornado para `gdb`. Em outras palavras, o comando `step` executa o programa *passo a passo*. O comando `next` é semelhante ao comando `step`; porém, a execução é interrompida somente quando a próxima linha está localizada na função corrente, ou seja, as funções invocadas não interrompem a execução. O comando `finish` executa o programa até o término da função corrente. Finalmente, o comando `continue` retoma a execução do programa.

C.3 Observação e Modificação do Estado do Programa

C.3.1 Observação

Uma das funções essenciais de um depurador simbólico é a inspeção do estado do programa. `Gdb` possibilita que o programador examine tanto a pilha de execução do programa (*call stack*) como os valores de variáveis, expressões e posições de memória.

O comando `frame` permite o exame da pilha de execução. Este comando descreve o estado do registro de ativação corrente apresentando ao programador o nome da função, os valores dos parâmetros passados na invocação e o código fonte e o número da linha do último comando executado. O comando `frame` possui variantes que permitem verificar as informações a respeito de qualquer registro de ativação que esteja na pilha de execução, assim como os valores dos parâmetros e variáveis locais contidos nestes registros. O comando `backtrace` apresenta todos os registros de ativação contidos na pilha de execução até o ponto onde a execução do programa foi interrompida.

No entanto, a maneira usual de examinar o estado do programa é usando o comando `print`. Este comando avalia e imprime os valores de uma expressão na linguagem em que o programa foi escrito. Qualquer tipo de constante, variável ou operador definido na linguagem pode fazer parte de uma expressão válida no comando `print`. Isto inclui expressões condicionais, chamadas de função e cadeias de caracteres. Para examinar a memória, independentemente do tipo dos dados associado a um endereço, existe o comando `x`.

C.3.2 Modificação

O depurador `gdb` permite que o programador altere o estado do programa para que ele possa confirmar uma hipótese sobre a localização ou a correção do defeito. Tipicamente, o programador altera o valor de variáveis e posições de memória utilizando o comando `set`. Por exemplo, o comando `set var i = 47` atribui o valor 47 à variável `i`. Analogamente, o comando `set` pode ser usado para atribuir um novo valor para um endereço de memória.

A própria seqüência de execução pode ser alterada pela modificação do estado do programa. `Gdb` possibilita que o *contador de programa* seja modificado por meio do comando `set`, de forma que a execução do programa é retomada em um ponto diferente.

C.4 Definição de Comandos pelo Usuário

`Gdb` permite que o usuário escreva *scripts* que constituem *novos* comandos incorporados ao depurador. Os *scripts* são escritos utilizando seqüências de comandos regulares de `gdb`, comandos de controle de fluxo e variáveis intermediárias.

Do ponto de vista de controle de fluxo, `gdb` fornece somente dois comandos básicos — *if* e *while*. Já as variáveis intermediárias — chamadas no jargão de `gdb` de *variáveis de conveniência* (*convenience variables*) — são sofisticadas, pois possuem características *dinâmicas*. Essas variáveis são criadas quando referenciadas pela primeira vez e assumem o tipo do objeto que é atribuído a elas. Outra característica importante é que elas não interferem com as variáveis do programa ou com sua execução. Todo nome precedido do caractere `$` é uma variável de conveniência. No Apêndice D são apresentados os *scripts* gerados automaticamente pela ferramenta `gdb/poke`.

Apêndice D

Scripts gerados pela gdb/poke

D.1 Scripts gerados pela gdb/poke

A seguir são apresentados os *scripts* gerados automaticamente pela gdb/poke quando o mantenedor ajusta a *adu* (65,(68,69),pa) para monitoração dos seus eventos de teste.

```
# GDB scripts for Testing Requirement <65,(68,69),{  pa }>

# Convenience variables initialization

set confirm off
delete
set confirm on
set $func_invoc = 0
set $nodef = 65
set $lastdef_invoc = -1
set $lastdef_seqexec = -1
set $lastinstance = -1
set $nosuc = 69
set $nopred = 68
set $instance = 0
set $last_node = -1

define cur_testreq
echo <65,(68,69),{  pa }>\n
end
```

```
define cur_exec
printf "Current instance %d, Function Invocation %d, node %d, sequence of execution %d\n",
               $instance,invoc_no,num_no,seq_exec
end

define resetvar
set $pred_visit = 0
set $active_instance = 0
set $found = 0
set $stopsearch = 0
set $func_invoc=0
set $last_node = -1
set $instance = 0
end

define rerun
reset
run
end


# Function entry point

sbreak testeprog.c:2080
commands
silent

set $pred_visit = 0
set $active_instance = 0
set $found = 0
set $stopsearch = 0
set $func_invoc=invoc_no
set $last_node = -1
while invoc_no < 5
cont
end
```

```
if $func_invoc == 1
  printf "Entered function cmp -- %dst invocation\n", $func_invoc
else
  if $func_invoc == 2
    printf "Entered function cmp -- %dnd invocation\n", $func_invoc
  else
    if $func_invoc == 3
      printf "Entered function cmp -- %drd invocation\n", $func_invoc
    else
      printf "Entered function cmp -- %dth invocation\n", $func_invoc
    end
  end
end

end

# Function exit points

sbreak testeprog.c:2459
commands
silent
end

# Redef nodes exit points

# Redef node 3

sbreak testeprog.c:2095
commands
silent
end
sbreak testeprog.c:2105
commands
silent
end

# Redef node 4
```

```
sbreak testeprog.c:2101
commands
silent
end

# Redef node 9

sbreak testeprog.c:2125
commands
silent
end

# Redef node 14

sbreak testeprog.c:2147
commands
silent
end

# Redef node 18

sbreak testeprog.c:2169
commands
silent
end

# Redef node 43

sbreak testeprog.c:2289
commands
silent
end
sbreak testeprog.c:2294
commands
silent
end
```

```
# Redef node 44

sbreak testeprog.c:2292
commands
silent
end

# Redef node 49

sbreak testeprog.c:2316
commands
silent
end
sbreak testeprog.c:2321
commands
silent
end

# Redef node 54

sbreak testeprog.c:2338
commands
silent
end
sbreak testeprog.c:2343
commands
silent
end

# Redef node 75

sbreak testeprog.c:2432
commands
silent
end
sbreak testeprog.c:2436
```

```
commands
silent
end

# def point -- exit points of node 65

sbreak testeprog.c:2384
commands
silent
end

# P-use point -- exit point of node  68

sbreak testeprog.c:2401
commands
silent
end
sbreak testeprog.c:2419
commands
silent
end

# P-use point -- entry point of node  69

sbreak testeprog.c:2403
commands
silent
end

# Set breakpoints for the predecessors of node 69, if any.



# next_check command

define next_check
next
```

```
if num_no != $last_node
  set $last_node = num_no
  if num_no == $nopred && $active_instance == 1
    set $pred_visit = 1
  else
    if num_no == $nosuc && $active_instance == 1 && $pred_visit == 1
      set $found = 1
      set $active_instance = 0
      set $instance = $instance+1
      set $pred_visit = 0
      printf "P-Use node has been reached\nUse value of p: "
      # Add the print command of the variable(s) of the dfa
      printf "Variable pa value: "
      output pa
      printf "\n"
    else
      if num_no ==  -1
        set $pred_visit = 0
      end
      if num_no == 82
        echo The exit of function has been reached and the use node was not found\n
        set $stopsearch = 1
        set $active_instance = 0
        set $pred_visit = 0
      else
        if num_no == 3 || num_no == 4 || num_no == 9 || num_no == 14 || num_no == 18
           || num_no == 43 || num_no == 44 || num_no == 49 || num_no == 54 || num_no == 75
          printf "A redefinition occurred at node %d\n", num_no
          set $active_instance = 0
          set $pred_visit = 0
        else
          if num_no == $nodef
            set $active_instance = 1
            set $pred_visit = 0
            set $lastdef_invoc = invoc_no
            set $lastdef_seqexec = seq_exec
            set $lastinstance = $instance
```

```
            printf "Def node has been hit\n"
            printf "Variable pa value: "
            output pa
            printf "\n"
          end
        end
      end
    end
  end
end
end


# next_puse command

define next_puse
set $found = 0
set $stopsearch = 0
while $found == 0 && $stopsearch == 0
  set $last_node = num_no
  if num_no == $nopred && $active_instance == 1
    set $pred_visit = 1
    cont
  else
    if num_no == $nosuc && $active_instance == 1 && $pred_visit == 1
      set $found = 1
      set $active_instance = 0
      set $instance = $instance+1
      set $pred_visit = 0
      list 2399,2399
      list 2403,2403
      printf "P-use node has been reached at node %d (%d,%d,%d)\n",
              num_no, num_no, invoc_no,seq_exec
      # Add the print command of the variable(s) of the dfa
      printf "Variable pa value: "
      output pa
      printf "\n"
```

```
  printf "Instance # %d\n",$instance
else
  if num_no ==  -1
    set $pred_visit = 0
  end
  if num_no == 82
    echo The exit of function has been reached and the use node was not found\n
    set $active_instance = 0
    set $pred_visit = 0
    cont
  else
    if num_no == 3 || num_no == 4 || num_no == 9 || num_no == 14 || num_no == 18
       || num_no == 43 || num_no == 44 || num_no == 49 || num_no == 54 || num_no == 75
      if num_no == 3
        list 2089,2093
        printf "Variable pa value: "
        output pa
        printf "\n"
        printf "A redefinition occurred at node %d (%d,%d,%d)\n",
                num_no,num_no,invoc_no,seq_exec
      else
        if num_no == 4
          list 2097,2100
          printf "Variable pa value: "
          output pa
          printf "\n"
          printf "A redefinition occurred at node %d (%d,%d,%d)\n",
                  num_no,num_no,invoc_no,seq_exec
        else
          if num_no == 9
            list 2124,2124
            printf "Variable pa value: "
            output pa
            printf "\n"
            printf "A redefinition occurred at node %d (%d,%d,%d)\n",
                   num_no,num_no,invoc_no,seq_exec
          else
```

```
if num_no == 14
  list 2146,2146
  printf "Variable pa value: "
  output pa
  printf "\n"
  printf "A redefinition occurred at node %d (%d,%d,%d)\n",
         num_no,num_no,invoc_no,seq_exec
else
  if num_no == 18
    list 2167,2168
    printf "Variable pa value: "
    output pa
    printf "\n"
    printf "A redefinition occurred at node %d (%d,%d,%d)\n",
           num_no,num_no,invoc_no,seq_exec
  else
    if num_no == 43
      list 2287,2287
      printf "Variable pa value: "
      output pa
      printf "\n"
      printf "A redefinition occurred at node %d (%d,%d,%d)\n",
             num_no,num_no,invoc_no,seq_exec
    else
      if num_no == 44
        list 2291,2291
        printf "Variable pa value: "
        output pa
        printf "\n"
        printf "A redefinition occurred at node %d (%d,%d,%d)\n",
               num_no,num_no,invoc_no,seq_exec
      else
        if num_no == 49
          list 2314,2314
          printf "Variable pa value: "
          output pa
          printf "\n"
```

```
                printf "A redefinition occurred at node %d (%d,%d,%d)\n",
                       num_no,num_no,invoc_no,seq_exec
              else
                if num_no == 54
                  list 2336,2336
                  printf "Variable pa value: "
                  output pa
                  printf "\n"
                  printf "A redefinition occurred at node %d (%d,%d,%d)\n",
                         num_no,num_no,invoc_no,seq_exec
                else
                  if num_no == 75
                    list 2430,2430
                    printf "Variable pa value: "
                    output pa
                    printf "\n"
                    printf "A redefinition occurred at node %d (%d,%d,%d)\n",
                           num_no,num_no,invoc_no,seq_exec
                  end
                end
              end
            end
          end
        end
      end
    end
  end

  set $active_instance = 0
  set $pred_visit = 0
else
  if num_no == $nodef
    set $active_instance = 1
    set $pred_visit = 0
    set $lastdef_invoc = invoc_no
    set $lastdef_seqexec = seq_exec
```

```
            set $lastinstance = $instance
            list 2383,2383
            printf "Def node has been hit at node %d (%d,%d,%d)\n",
                    num_no, num_no, invoc_no,seq_exec
            printf "Variable pa value: "
            output pa
            printf "\n"
          end
        end
      cont
      end
    end
  end
end
end

define next_use
next_puse
end


# Commands to access testing requirements instances

define firstdef
if invoc_no <= 5
  if invoc_no == 5
    if seq_exec <= 25
      if seq_exec == 25
        list 2383,2383
        printf "First def instance has been reached at node %d (%d,%d,%d)\n",
               num_no, num_no,invoc_no,seq_exec
        printf "Variable pa value: "
        output pa
        printf "\n"
        set $active_instance = 1
        set $last_node = num_no
      else
```

```
        while seq_exec != 25 || invoc_no != 5
          cont
        end
        if seq_exec == 25 && invoc_no == 5
          set $active_instance = 1
          set $last_node = num_no
          list 2383,2383
          printf "First def instance has been reached at node %d (%d,%d,%d)\n",
                  num_no, num_no,invoc_no,seq_exec
          printf "Variable pa value: "
          output pa
          printf "\n"
        else
          echo There is no first def for this test case.\n
        end
      end
    else
      echo First def instance has already been reached\n
    end
  else
    while seq_exec != 25 || invoc_no != 5
      cont
    end
    if seq_exec == 25 && invoc_no == 5
      set $active_instance = 1
      set $last_node = num_no
      list 2383,2383
      printf "First def instance has been reached at node %d (%d,%d,%d)\n",
              num_no, num_no,invoc_no,seq_exec
      printf "Variable pa value: "
      output pa
      printf "\n"
    else
      echo There is no first def for this test case.\n
    end
  end
else
```

```
  echo First def instance has already been reached\n"
end
end

define instance_def
if invoc_no <= $lastdef_invoc
  if invoc_no == $lastdef_invoc
    if seq_exec <= $lastdef_seqexec
      if seq_exec == $lastdef_seqexec
        list 2383,2383
        printf "Last instance Def has been reached at node %d (%d,%d,%d)\n",
               num_no, num_no,invoc_no,seq_exec
        printf "Variable pa value: "
        output pa
        printf "\n"
        set $instance = $lastinstance
        set $active_instance = 1
        set $last_node = num_no
      else
        while seq_exec != $lastdef_seqexec || invoc_no != $lastdef_invoc
          cont
        end
        if seq_exec == $lastdef_seqexec && invoc_no == $lastdef_invoc
          set $active_instance = 1
          set $last_node = num_no
          set $instance = $lastinstance
          list 2383,2383
          printf "Last instance def has been reached at node %d (%d,%d,%d)\n",
                 num_no, num_no,invoc_no,seq_exec
          printf "Variable pa value: "
          output pa
          printf "\n"
        else
          echo There is no last instance def for this test case.\n
        end
      end
    else
```

```
      echo Last instance def has already been reached\n
    end
  else
    while seq_exec != $lastdef_seqexec || invoc_no != $lastdef_invoc
      cont
    end
    if seq_exec == $lastdef_seqexec && invoc_no == $lastdef_invoc
      set $active_instance = 1
      set $last_node = num_no
      set $instance = $lastinstance
      list 2383,2383
      printf "Last instance def has been reached at node %d (%d,%d,%d)\n",
             num_no, num_no,invoc_no,seq_exec
      printf "Variable pa value: "
      output pa
      printf "\n"
    else
      echo There is no last instance def for this test case.\n
    end
  end
else
  echo Last instance def has already been reached\n"
end
end

define lastdef
if invoc_no <= 1710
  if invoc_no == 1710
    if seq_exec <= 37
      if seq_exec == 37
        list 2383,2383
        printf "Last def instance has been reached at node %d (%d,%d,%d)\n",
               num_no, num_no,invoc_no,seq_exec
        printf "Variable pa value: "
        output pa
        printf "\n"
        set $instance = 110
```

```
        set $active_instance = 1
        set $last_node = num_no
      else
        while seq_exec != 37 || invoc_no != 1710
          cont
        end
        if seq_exec == 37 && invoc_no == 1710
          list 2383,2383
          set $instance = 110
          set $active_instance = 1
          set $last_node = num_no
          printf "Last def instance has been reached at node %d (%d,%d,%d)\n",
                  num_no, num_no,invoc_no,seq_exec
          printf "Variable pa value: "
          output pa
          printf "\n"
        else
          echo There is no last def for this test case.\n
        end
      end
    else
      echo Last def instance has already been reached\n
    end
  else
     while seq_exec != 37 || invoc_no != 1710
       cont
     end
     if seq_exec == 37 && invoc_no == 1710
          list 2383,2383
       set $instance = 110
        set $active_instance = 1
        set $last_node = num_no
       printf "Last def instance has been reached at \
node %d (%d,%d,%d)\n",num_no, num_no,invoc_no,seq_exec
        printf "Variable pa value: "
        output pa
        printf "\n"
```

```
    else
      echo There is no last def for this test case.\n
    end
  end
else
  echo Last def instance has already been reached\n
end
end


define instance_puse
set $instance_value = $arg0
set $instance_max = 111
if $instance_value <= $instance_max && $instance_value > 0
  if $instance < $instance_value
    while $instance < $instance_value
    # goto use without stopping in the exit of the functions
      next_puse
    end
    if $instance != $instance_value
      echo There is no such use instance.\n
    else
      printf "Use instance %d has been reached\n",$instance
    end
  else
    if $instance == $instance_value
      printf "Use instance %d has been reached\n",$instance
    else
    printf "Instance # %d has already been passed\n",$instance
    end
  end
else
  printf "Instance value %d greater than the maximum number of instances (%d)\n",
         $instance_value,$instance_max
end
end
```

```
define instance_use
instance_puse $arg0
end
```

Apêndice E

Ramos Essenciais Estendidos na Presença de Caminhos Não-executáveis

Neste apêndice são descritas as alterações nos algoritmos da POKE-TOOL que determinam as *adpu*s definidas através de ramos essenciais estendidos. As alterações realizadas visam garantir que estas associações possam ser usadas na determinação de conjuntos adequados aos critérios potenciais usos na presença de caminhos não-executáveis.

E.1 Ramos Essenciais e Ramos Essenciais Estendidos

Uma das aplicações do conceito de *ramo essencial* introduzido por Chusho (1987) é na redução do custo da análise de adequação do critério todos os ramos. O conjunto de ramos essenciais[1] é um subconjunto dos ramos do programa que possui a seguinte propriedade — um conjunto de casos de teste que exercita os ramos essenciais exercita todos os ramos do programa. Esta propriedade permite reduzir o custo da análise de adequação visto que apenas uma parcela dos ramos do programa precisa ser monitorada.

Maldonado (Maldonado, 1991; Maldonado et al., 1992b) estendeu o uso dos ramos essenciais para o contexto de fluxo de dados, mais especificamente, para aplicação na análise de adequação dos critérios potenciais usos. Os *ramos essenciais estendidos de fluxo de dados* visam garantir a execução de todos os ramos dos *grafos(i)*. Estes grafos são determinados a partir dos nós do grafo de fluxo de controle que possuem definições de variáveis (chamados nós i) e abstraem caminhos que exercitam os requisitos dos critérios potenciais usos.

A idéia é que, exercitando-se os ramos essenciais estendidos presentes em um grafo(i), garante-se que todos os requisitos dos critérios potenciais usos, que começam no nó i, são exercitados. A partir

[1] Na verdade, um programa pode possuir diferentes conjuntos de ramos essenciais no quais o número de ramos é mínimo. Chusho (1987) apresenta um algoritmo para determinar um desses conjuntos; por isso, é feita referência a*o* conjunto de ramos essenciais do programa.

dessa propriedade, os ramos essenciais estendidos foram utilizados para estabelecer *adpu*s cujo exercício *implica* (*inclui*) o exercício de várias outras, reduzindo o número de associações requeridas e, assim, reduzindo o custo a análise de adequação dos critérios potenciais usos.

Entretanto, observou-se que esta propriedade pode não ser válida na presença de caminhos não-executáveis para o critério todos os potenciais usos. Como conseqüência, *adpu*s *executáveis*, exigidas de acordo com a definição original desse critério, podem não ser exercitadas por um conjunto de casos de teste adequado às *adpu*s baseadas em ramos essenciais estendidos.

Para que esta situação não ocorra, alterou-se a maneira de determinar os grafos(i) e essas *adpu*s, de forma a garantir que as novas *adpu*s baseadas em ramos essenciais estendidos possam ser usadas na determinação de conjuntos adequados aos critérios potenciais usos na presença de caminhos não-executáveis. A seguir, é descrita a solução para este problema.

E.2 Grafo(i)

Seja DEF o conjunto de nós do grafo (G,N,R,e,s) de um procedimento P definido da seguinte maneira: DEF = $\{\ i \in N$ tal que $defg(i) \neq \phi$ onde $defg(i)$ é o conjunto de variáveis definidas em $i\}$. A partir de cada nó $i \in$ DEF, é possível derivar um grafo que inclui todos os potenciais du-caminhos originados em i. Este grafo é obtido através de uma busca em profundidade em G e é chamado de *grafo(i)*. Um nó $k \in N$ é incluído em um grafo(i), e referenciado como nó k_q, $q \geq 1$, se existir pelo menos um potencial du-caminho $\pi_q = i,\ldots,k$ c.r.a. uma das variáveis definidas em i. Os nós k_q do grafo(i) são caracterizados por duas informações: o nó $k \in N$ e o conjunto $deff(k_q) \subseteq defg(i)$. O conjunto $deff(k_q)$ contém as variáveis definidas em i que estão vivas em k_q por qualquer caminho do grafo(i) de i até k_q. Como podem existir vários potenciais du-caminhos $\pi_1, \pi_2, \ldots, \pi_q$ de i até k, tal que $deff(k_1) \neq deff(k_2) \neq \ldots \neq deff(k_q)$, um nó $k \in N$ pode possuir diferentes imagens $k_1, k_2, \ldots, k_q$ no grafo(i). Considere-se o programa exemplo 4 contido na Figura E.1 e o seu grafo de fluxo de controle, apresentado na Figura E.2, no qual as variáveis definidas em cada nó i estão indicadas. A Figura E.3 contém o grafo(1) obtido a partir desse grafo de fluxo de controle, bem como os conjuntos $deff$ das imagens do nó 2.

E.3 Ramos Essenciais Estendidos

Os ramos essenciais estendidos são ramos (k_q,l_q) do grafo(i) selecionados de forma a garantir que todo caminho do grafo(i) contenha pelo menos um ramo essencial. Eles são determinados a partir de duas fontes: o algoritmo modificado de Chusho e o algoritmo de construção dos grafos(i) (Maldonado, 1991). Por isso, existem dois tipos de ramos essenciais estendidos de fluxo de dados. Um ramo (k_q,l_q) do grafo(i) é um ramo *essencial estendido* p se o ramo (k,l) foi identificado como essencial a partir da aplicação do algoritmo modificado de Chusho no grafo G do procedimento. Adicionalmente,

```
void main()
/*  1 */        {
/*  1 */           int  a,b,c,d;
/*  1 */           a = b = c = d = 1;
/*  2 */           while(a < 10)
/*  3 */           {
/*  3 */              if(c != 1)
/*  4 */              {
/*  4 */                 if(c>d)
/*  5 */                 {
/*  5 */                    scanf("%d", &a);
/*  5 */                    ++a;
/*  5 */                    scanf("%d", &b);
/*  5 */                    ++b;
/*  5 */                 }
/*  6 */                 else
/*  6 */                    printf("%d %d\n", c, d);
/*  7 */                 if(a > b)
/*  8 */                    printf("%d\n", a);
/*  9 */                 else
/*  9 */                    printf("%d\n", b);
/* 10 */              }
/* 11 */              else
/* 11 */              {
/* 11 */                 scanf("%d", &c);
/* 11 */                 ++c;
/* 11 */                 scanf("%d", &d);
/* 11 */                 ++d;
/* 11 */              }
/* 12 */           }
/* 13 */        }
```

Figura E.1: Programa exemplo 4.

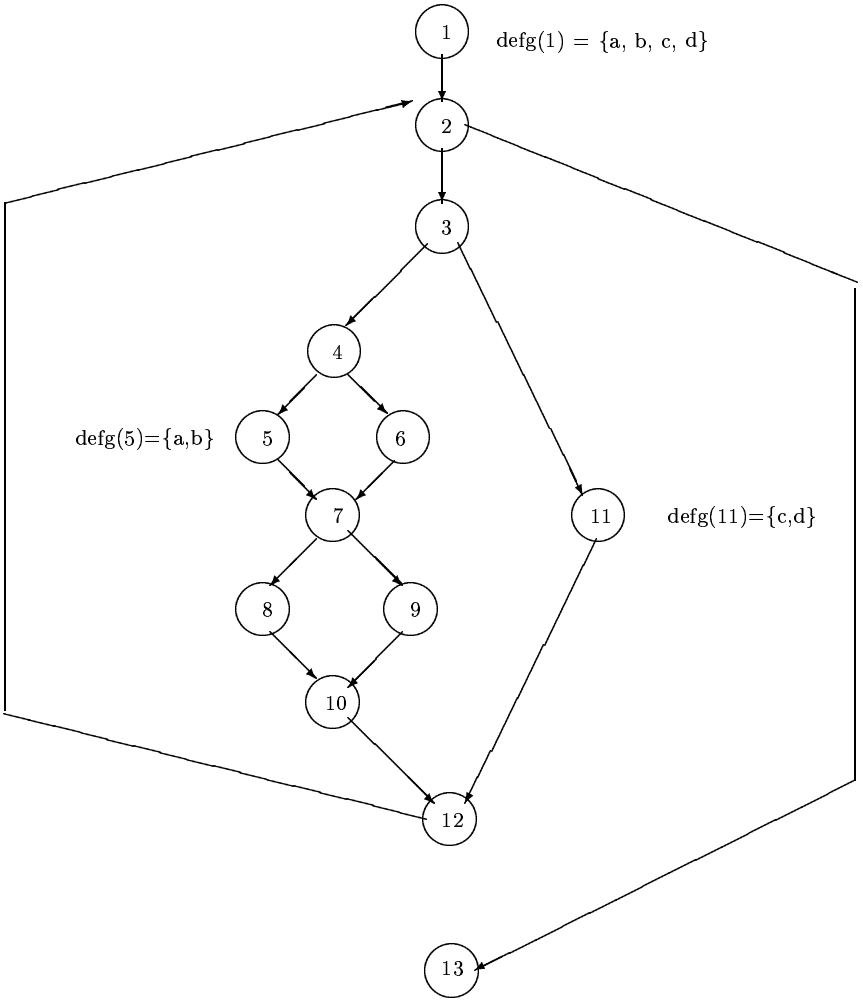


Figura E.2: Grafo de fluxo de controle do programa exemplo 4.

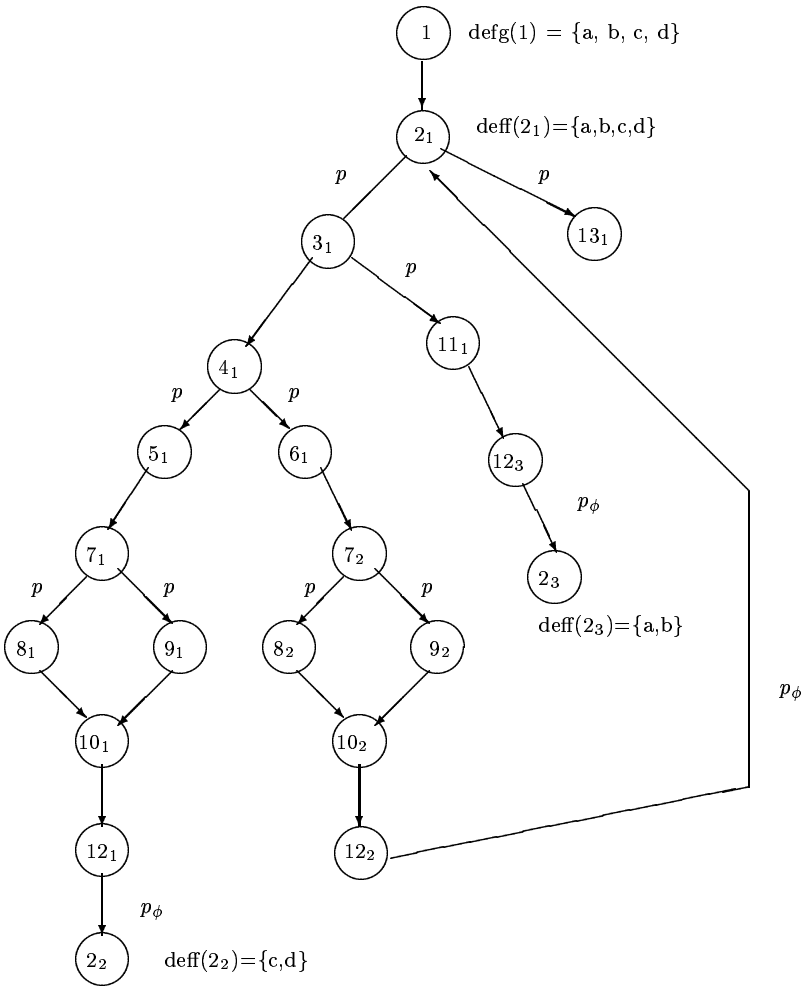


Figura E.3: Grafo(1) contendo potenciais du-caminhos.

foi introduzido por Maldonado (1991) o conceito de ramo essencial p_ϕ obtido durante a construção do grafo(i). Um ramo (k_q, l_q) do grafo(i) é um ramo *essencial estendido* p_ϕ se existe um caminho $\pi_q = (i, \ldots, k, l)$ e:

i) $deff(l_q) = \phi$ no caminho π_q, ou seja, todas as variáveis definidas no nó i foram redefinidas no caminho π_q=$i, \ldots, k, l$; ou

ii) O nó l já faz parte do caminho π_q=$i, \ldots, k, l$, ou seja, l_q é a segunda imagem de $l \in N$ no caminho π_q do grafo(i).

Chaim (1991) descreve a implementação do algoritmo modificado de Chusho e do algoritmo para a obtenção dos grafo(i). Na Figura E.3, estão indicados os ramos essenciais estendidos do grafo(1).

E.4 Adpus baseadas em Ramos Essenciais Estendidos

Os grafos(i) são construídos de tal forma que a execução de pelo menos um dos caminhos entre i e k_q implica o exercício dos requisitos dos critérios potenciais usos do nó i até o nó k c.r.a. variáveis do conjunto $deff(k_q)$. Portanto, o grafo(i) constitui uma abstração dos requisitos dos critérios potenciais usos com relação às variáveis definidas no nó i. Assim, semelhantemente aos ramos essenciais de Chusho (1987) para o critério todos os ramos, a execução dos ramos essenciais estendidos dos grafo(i) implica o exercício dos requisitos dos critérios potenciais usos.

Dessa maneira, os requisitos de teste para os critérios potenciais usos foram redefinidos a partir dos grafos(i) como as associações (i,(k,l),D) tal que (k,l) pertence ao conjunto R de ramos do programa, (k_q,l_q) pertence ao conjunto de ramos essenciais estendidos do grafo(i) e D é igual a $deff(k_q)$ (Maldonado, 1991). Note-se que é possível derivar conjuntos adequados aos critérios todos os potenciais usos e todos os potenciais usos/du a partir das *adpu*s baseadas em ramos essenciais estendidos. O conjunto de caso de teste que exercita as *adpu*s (i,(k,l),D) através de caminhos livres de definição c.r.a. todas as variáveis do conjunto D ($deff(k_q)$) é adequado ao critério todos os potenciais usos. Analogamente, para o critério todos os potenciais usos/du, os conjuntos de casos de teste devem exercitar as mesmas associações, porém, utilizando potenciais du-caminhos c.r.a. às variáveis do conjunto D. Abaixo são apresentadas as *adpu*s derivadas diretamente dos ramos essenciais estendidos do grafo(1) apresentado na Figura E.3:

```
Associacoes requeridas pelo Grafo( 1)

    1) <1,(2,13),{ a, b, c, d }>
    2) <1,(3,11),{ a, b, c, d }>
    3) <1,(4,6),{ a, b, c, d }>
    4) <1,(7,9),{ a, b, c, d }>
```

```
5) <1,(7,9),{ c, d }>
6) <1,(12,2),{ a, b }>
7) <1,(12,2),{ a, b, c, d }>
8) <1,(12,2),{ c, d }>
9) <1,(7,8),{ a, b, c, d }>
10) <1,(7,8),{ c, d }>
11) <1,(4,5),{ a, b, c, d }>
```

No entanto, observou-se que as *adpu*s definidas usando ramos essenciais estendidos diretamente dos grafo(i) garantem a exercício dos requisitos dos critérios potenciais usos desde que essas associações sejam executáveis. Porém, quando caminhos não-executáveis são considerados, esta propriedade é válida somente para o critério todos os potenciais usos/du; para o critério todos os potenciais usos, há situações em que ela *não* é válida.

Para exemplificar essas situações, considere-se o programa exemplo 4 da Figura E.1 e a associação baseada em ramos essenciais estendidos `(1,(2,13),{ a, b, c, d })`. O exercício desta associação implica o exercício das seguintes *adpu*s exigidas pela definição original do critério todos os potenciais usos contida na Subseção 2.3.1 (de agora em diante essas associações são referenciadas como *associações originais*): `(1,(2,13),a)`, `(1,(2,13),b)`, `(1,(2,13),c)` e `(1,(2,13),d)`. Entretanto, de acordo com as condições de executabilidade do programa, a associação `(1,(2,13),{ a, b, c, d })` é não-executável, pois o nó 11 é sempre executado antes do ramo (2,13), o que provoca a redefinição das variáveis `c` e `d`. Observe-se, porém, que as associações originais `(1,(2,13),a)` e `(1,(2,13),b)`, que deveriam ser *incluídas* pela *adpu* `(1,(2,13),{ a, b, c, d })`, são executáveis. Analogamente, as associações `(1,(7,9),{ a, b, c, d })` e `(1,(7,9),{ c, d })` são igualmente não-executáveis porque o nó 11 é executado previamente ao ramo (7,9); todavia, existem caminhos executáveis que exercitam as associações originais `(1,(7,9),a)` e `(1,(7,9),b)`.

Este fato significa que é possível criar um conjunto T de casos de teste *adequado* às *adpu*s baseadas em ramos essenciais estendidos, mas *não-adequado* à definição do critério todos os potenciais usos, pois não há garantia de que as *adpu*s originais executáveis serão exercitadas pelos casos de teste de T. Note-se, todavia, que este problema não ocorre para os critérios todos os potenciais usos/du, pois, neste caso, as associações originais são também não-executáveis.

E.5 Adpus baseadas em Ramos Essenciais Estendidos na Presença de Caminhos Não-executáveis

Um grafo(i), como descrito na Subseção E.2, contém todos os potenciais du-caminhos entre o nó i e o nó k. A Figura E.3 apresenta o grafo(1) do programa exemplo que abstrai potenciais du-caminhos. Por isso, eles são uma abstração dos requisitos dos critérios todos os potenciais usos/du, mesmo na

presença de caminhos não-executáveis. Para que isto seja válido para os requisitos do critério todos os potenciais usos, é necessário que os grafos(i) contenham todos os *caminhos livres de definição* entre o nó i e um nó k.

Para superar esta limitação, os grafos(i) foram alterados para conter caminhos livres de definição, de tal forma que um nó $k \in N$ é incluído no grafo(i), e referenciado como nó k_q, $q \geq 1$, se existir pelo menos um *caminho livre de definição* $\pi_q = i,\ldots,k$ c.r.a. uma das variáveis definidas em i. Assim, as novas *adpu*s são derivadas a partir dos ramos essenciais estendidos dos novos grafos(i) da mesma forma; porém, os conjuntos de variáveis definidas no nó i associados às novas *adpu*s são estabelecidos de maneira diferente:

- se existe apenas uma imagem (k_1,l_1) do ramo essencial estendido, então é gerada uma *adpu* (i,(k,l),D) onde $D = deff(k_1)$;
- se (k_1,l_1), (k_2,l_2), ..., (k_q,l_q), $q > 1$, são ramos essenciais estendidos do grafo(i), então a partir destes ramos são derivadas as *adpu*s (i,(k,l),D_m), onde $0 < m \leq q$ e os conjuntos D_m são subconjuntos maximais de algum $deff(k_q)$ tal que $D_1 \bigcap \ldots \bigcap D_m = \phi$ e $D_1 \bigcup D_2 \ldots \bigcup D_m = deff(k_1) \bigcup deff(k_2) \ldots \bigcup deff(k_q)$.

Considere-se o novo grafo(i) descrito na Figura E.4 obtido do programa da Figura E.1. As novas *adpu*s baseadas nos ramos essenciais estendidos do novo grafo(1) da Figura E.4 são apresentadas a seguir:

```
Associacoes requeridas pelo Grafo( 1)
      1) <1,(7,9),{ c, d }>
      2) <1,(7,9),{ a, b }>
      3) <1,(2,13),{ c, d }>
      4) <1,(2,13),{ a, b }>
      5) <1,(3,11),{ c, d }>
      6) <1,(3,11),{ a, b }>
      7) <1,(4,6),{ c, d }>
      8) <1,(4,6),{ a, b }>
      9) <1,(7,8),{ c, d }>
     10) <1,(7,8),{ a, b }>
     11) <1,(4,5),{ c, d }>
     12) <1,(4,5),{ a, b }>
     13) <1,(12,2),{ c, d }>
     14) <1,(12,2),{ a, b }>
```

Observe-se que no novo grafo(i) o ramo (2,13) possui três imagens — (2_1,13_1), (2_2,13_2) e (2_3,13_3), enquanto que no grafo(i) anterior apenas uma — (2_1,13_1). Isto ocorre porque o novo grafo(i) abstrai

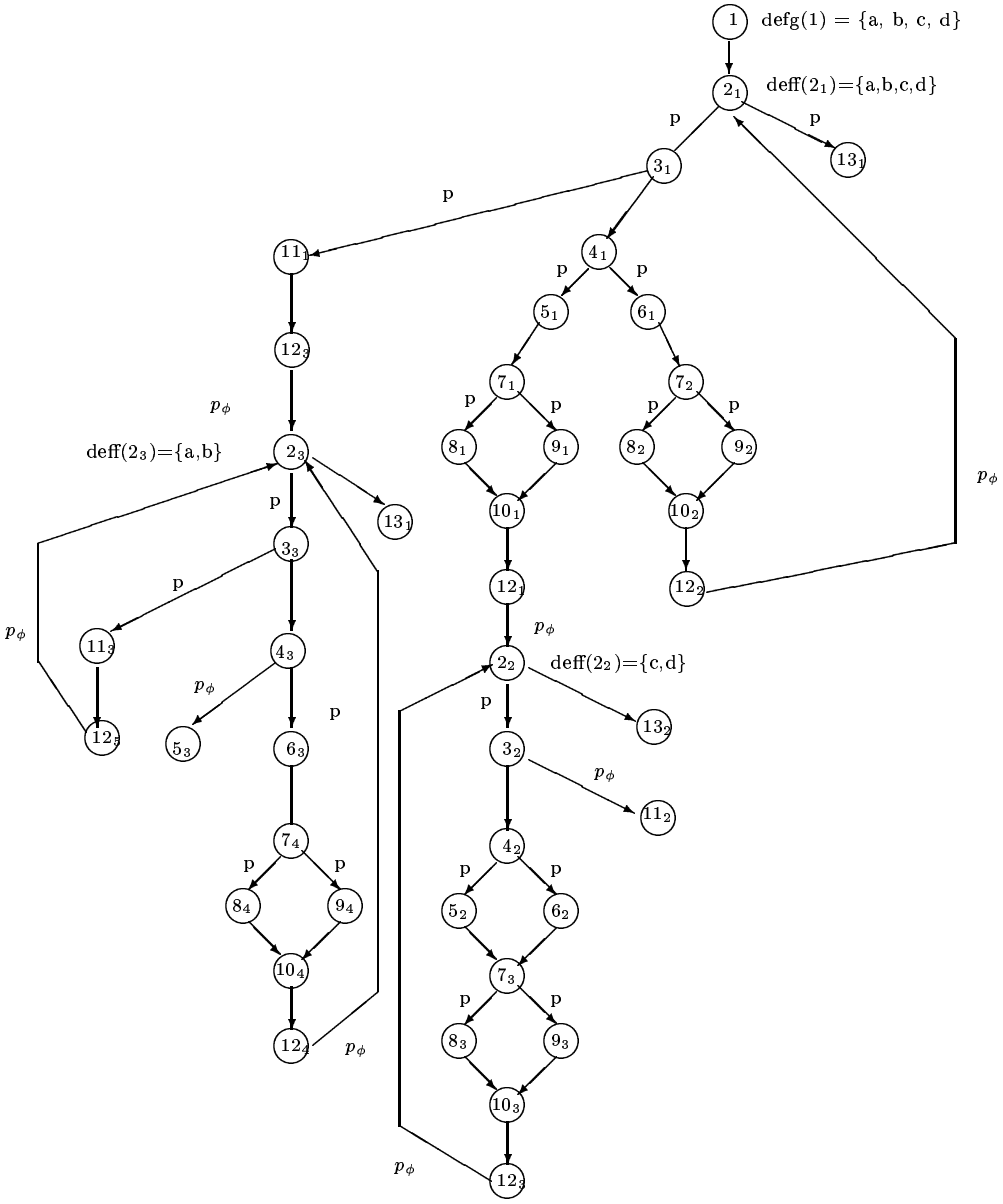


Figura E.4: Grafo(1) contendo caminhos livres de definição.

caminhos livres de definição e não potenciais du-caminhos. As novas *adpu*s para esses ramos essenciais estendidos são (1,(2,13),{ a, b }) e (1,(2,13),{ c, d }), onde $D_1 =$ { a, b } e $D_2 =$ { c, d } tal que $D_1 \subseteq deff(2_3)$, $D_2 \subseteq deff(2_2)$, $D_1 \bigcap D_2 = \phi$ e $deff(2_1) \bigcup deff(2_2) \bigcup deff(2_3) =$ { a, b, c, d } = { a, b } $\bigcup$ { c, d } $= D_1 \bigcup D_2$.

É interessante observar que as novas *adpu*s (i, (k,l),D_m) possuem a seguinte propriedade: se as variáveis $v, v' \in D_m$, então todo caminho que é livre de definição de i até (k,l) com respeito à variável v também o é com respeito a v'. Esta propriedade decorre do fato de os conjuntos D_m serem disjuntos, mas contidos em algum $deff(k_q)$; além disso, para que nenhuma variável pertencente aos conjuntos $deff(k_q)$ deixe de ser exercitada, a união desses conjuntos é igual à união dos $deff(k_q)$.

Segundo esta nova abordagem, a *adpu antiga* (1,(2,13),{ a, b, c, d }) é substituída por duas novas *adpu*s: (1,(2,13),{ a, b }) e (1,(2,13),{ c, d }). Assim, a *adpu* (1,(2,13),{ a, b }) é executável e o seu exercício implica que as associações originais (1,(2,13),a) e (1,(2,13),b) também foram exercitadas. A *adpu* (1,(2,13),{ c, d }) é não-executável, bem como as associações originais (1,(2,13),d) e (1,(2,13),c), que seriam incluídas por esta associação. Assim, pode-se observar que os conjuntos adequados às novas *adpu*s são também adequados ao critério todos os potenciais usos.

Observe-se ainda que conjuntos adequados aos critérios todos os potenciais usos/du podem ser selecionados através das novas *adpu*s, mesmo na presença de caminhos não executáveis. Isto ocorre porque os potenciais du-caminhos são também caminhos livres de definição e, portanto, estão representados nos *novos* grafos(i). A única diferença é que associações adicionais podem ser exigidas. Por exemplo, a *adpu* (1,(2,13),{ a, b, c, d }) é representada por duas *adpu*s, embora isto não seja necessário para critério todos os potenciais usos/du.

E.6 Alterações na Implementação da Ferramenta POKE-TOOL

As mudanças na ferramenta POKE-TOOL ficaram restritas ao módulo de geração de descritores (**descr**). Este módulo é responsável por determinar os requisitos de teste e os autômatos (descritores) utilizados na análise de adequação (veja Subseção 3.2.4). Para os critérios potenciais usos, o módulo **descr** determina os grafos(i) e as associações baseadas em ramos essenciais estendidos; daí a necessidade de alterá-lo para incluir a geração dos novos grafos(i) e das novas *adpu*s. Note-se, todavia, que a representação das novas *adpu*s (e dos seus descritores) é idêntica à utilizada anteriormente na POKE-TOOL; por isso, não houve necessidade de modificar os demais módulos da ferramenta. As alterações realizadas no módulo **descr** foram as seguintes:

- No algoritmo de construção dos grafos(i), a busca em profundidade foi alterada para determinar caminhos livres de definição. Neste mesmo algoritmo, a determinação dos ramos essenciais p_ϕ foi ligeiramente alterada devido ao fato dos novos grafos(i) conterem caminhos livres de definição.

A condição **ii)** para caracterização de um ramo p_ϕ foi reescrita da seguinte maneira:

ii) O nó l já faz parte do caminho $\pi_q{=}i, \ldots, k, l$ e l é dominador de k.

Ambas as alterações requereram mudanças mínimas neste algoritmo cuja nova versão está descrita no na Seção E.6.1

- A geração das *adpu*s baseadas em ramos essenciais estendidos foi modificada para incluir a determinação dos conjuntos D_m. Estes conjuntos são obtidos a partir dos $def f(k_1), \ldots, def f(k_q)$ através de um algoritmo na Seção E.6.2.

E.6.1 Algoritmo para Geração dos Grafos(i)

As modificações no algoritmo de geração dos grafos(i) da POKE-TOOL são apresentadas a seguir. As estruturas de dados não sofreram alterações mas são descritas abaixo para possibilitar o entendimento do algoritmo. As alterações propriamente ditas estão impressas em **negrito** para diferenciar do fonte `máquina de escrever` utilizado para descrever o texto original do algoritmo.

Estrutura de Dados

A estrutura de dados utilizada para implementar os grafo(i) é uma lista ligada de nós, onde cada elemento dessa lista é um nó do grafo(i). Os elementos dessa primeira lista contém um campo que é um apontador para uma outra lista ligada que representa os sucessores do nó do grafo(i). Dessa maneira, o grafo(i) é representado através de várias listas; essa solução foi adotada porque, *a priori*, não sabemos quantos elementos podem existir no grafo(i) e esse número pode exceder o número de nós do grafo de fluxo de controle (GFC).

A primeira lista é composta de elementos do tipo NOGRAFOI; os elementos desse tipo são compostos dos seguintes campos:

num_grafo_i : número de identificação desse nó no grafo(i);

infosuc : este campo contém as informações a respeito do nó do grafo(i), é do tipo INFODFNO (a ser explicado mais adiante);

repetido : campo que indica se esse nó do grafo(i) tem o mesmo número no GFC de outro nó do grafo(i);

sucgrfi : apontador para os sucessores do nó no grafo(i); e

next : endereço do próximo elemento.

O campo INFODFNO contém as informações sobre o nó do grafo(i), que são:

num_no_G : o número no GFC do nó do grafo(i); e

deff : o conjunto deff desse nó do grafo(i).

Observe-se que essas duas informações identificam unicamente um nó do grafo(i).

Voltando aos elementos da lista de nós do grafo(i), notamos que o campo **sucgfi** é o apontador para os sucessores do nó. Este campo aponta para uma lista de elementos do tipo SUCGRAFOI; este tipo descreve as informações relativas a um sucessor de um nó do grafo(i). Abaixo, são apresentados os campos desse tipo:

num_grafo_i : idêntico ao campo do tipo NOGRAFOI;

tipo : este campo identifica se o ramo constituído pelo nó do grafo(i) e o seu sucessor é do tipo essencial estendido (p ou p_ϕ) ou *herdeiro*, isto é, não é essencial;

no_address : este campo contém o endereço do nó sucessor na lista de nós do grafo(i) (lista de NOGRAFOI's); e

next : endereço do próximo sucessor.

Logo, o grafo(i) é representado pela lista de elementos do tipo NOGRAFOI, que são os nós do grafo(i) propriamente ditos, e por listas de sucessores (SUCGRAFOI's) associadas a cada elemento da lista de NOGRAFOI's.

Durante o desenvolvimento de um grafo(i) são obtidas todas as informações para a criação dos autômatos finitos (chamados descritores) associados aos critérios Potenciais Usos. Essas informações estão congregadas em uma estrutura de dados chamada **pdes**. **pdes** é um vetor cujos elementos são do tipo INFODESCRITORES; cada elemento desse vetor congrega as informações obtidas durante o desenvolvimento de um grafo(i). As informações contidas em cada elemento de **pdes** são as seguintes:

Ni : conjunto de "bits" que contém os nós do GFC que fazem parte do grafo(i);

Nt : conjunto de "bits" que contém os nós do GFC que são nós terminais do grafo(i);

grafo_i : apontador para a lista de nós do grafo(i), ou seja, para o grafo(i);

prim_ramos : apontador para a lista de ramos essenciais estendidos que fazem parte do grafo(i).

O campo **prim_ramos** aponta para uma lista de ramos do GFC que são essenciais estendidos no grafo(i). Um ramo pode ocorrer mais de uma vez em um grafo(i), ou melhor, possuir várias imagens no grafo(i) (ver Subseção E.2), pois é comum ocorrer a duplicação de partes do GFC quando é criado um grafo(i). As diferentes ocorrências de um ramo no grafo(i) indicam que, em termos de fluxo de dados, esses ramos são diferentes e devem ser exercitados pelos casos de teste para a satisfação dos critérios todos os potenciais usos e todos os potenciais usos/du. Para diferenciar entre as várias ocorrências de

um ramo essencial estendido no grafo(i) é utilizado o conjunto *deff* do nó fonte de cada ocorrência do ramo.

À medida que é construído um grafo(i), são colocados na lista apontada por **prim_ramos** todos os ramos (*i,j*) essenciais estendido, bem como os conjuntos *deff* de cada ocorrência desses ramos. **prim_ramos** aponta para registros do tipo ARCPRIM; ARCPRIM é definido da seguinte maneira:

ramo : é um campo do tipo PARINT; PAIRINT é um registro que contém um par de inteiros definidos nos campos **abs** e **coor**. O nó fonte do ramo é armazenado em **abs** e o nó alvo em **coor**;

deff_ptr : é o apontador para uma lista cujos elementos são conjuntos de "bits"; essa lista contém os *deff* das várias ocorrências de um ramo *(i,j)*; e

next : aponta para o próximo elemento.

Algoritmo de Geração dos Grafo(i)

As estruturas de dados apresentadas acima são utilizadas pela POKE-TOOL. Antes de iniciar a descrição do algoritmo de geração dos grafo(i), vamos introduzir algumas estruturas auxiliares utilizadas nesse algoritmo.

Uma delas é o tipo de dado LIST, que nada mais é do que uma lista de elementos cujo conteúdo é do tipo INFODFNO. Mais precisamente, os campos de um elemento do tipo LIST são:

sucessor : campo do tipo INFODFNO; e

next : próximo elemento da lista.

Outra estrutura auxiliar é do tipo de dado ELEMSUC; os elementos desse tipo são definidos através dos campos:

infosuc : campo do tipo INFODFNO; e

next : apontador para uma lista de elementos do tipo LIST.

O Algoritmo

ger_grafo_i (g_suc: GRAFO; **var** pdes: INFODESCRITORES)

ENTRADAS:

g_suc : apontador para o GFC do programa.

SAÍDA:

pdes :vetor com as informações obtidas da geração do grafo(i).

VARIÁVEIS LOCAIS:

conodef : conjunto de nós i do GFC tal que $defg(i) \neq \phi$.

pil_cam : pilha que armazena o caminho percorrido no GFC a partir do nó i onde $defg(i) \neq \phi$. Esta pilha contém os números dos nós do GFC pertencentes ao caminho sendo percorrido.

conj_suc : pilha que auxilia no percorrimento em profundidade do GFC; seus elementos são do tipo ELEMSUC.

x : variável do tipo inteiro que contém os números dos nós i retirados de **conodef**.

deff : conjunto de "bits" que contém o conjunto de variáveis definidas em i mas não redefinidas no particular caminho sendo percorrido pelo algoritmo.

ref : variável do tipo ELEMSUC que armazena informações do nó i que dá origem ao grafo(i).

elemento : variável do tipo ELEMSUC que armazena os elementos retirados de **conjsuc**.

nosucg : variível do tipo INFODFNO que armazena informações sobre o sucessor de **elemento** que está sendo inserido no grafo(i).

elem_aux : variável do tipo ELEMSUC auxiliar.

i : contador de grafo(i).

fonte : variável inteira que contém o número no GFC do nó fonte do ramo que será inserido no grafo(i).

alvo : variável inteira que contém o número no GFC do nó alvo do ramo que será inserido no grafo(i).

VARIÁVEIS GLOBAIS:

info_no : vetor que contém, entre outras coisas, os conjuntos associados a cada nó do GFC.

FUNÇÕES e PROCEDIMENTOS chamados

det_conodef(conodef) : determina o conjunto de nós i do GFC tal que $defg(i) \neq \phi$.

ret_elem(conodef) : retira um elemento do conjunto **conodef**.

ini_info_descritores(pdes[i]) : inicia o elemento i do vetor **pdes**. Essa iniciação consiste em "resetar" os conjuntos de "bits" **pdes[i].Ni** e **pdes[i].Nt** e ajustar os campos **pdes[i].grafo_i** e **pdes[i].prim_ramos** com o valor **nil**.

grfi_inicializacao(pdes[i], ref) : esta função insere no grafo(i) apontado por **pdes[i].grafo_i** o nó definido por **ref.infosuc.num_no_G** e **ref.infosuc.deff** que nada mais é do que o nó *i* e o conjunto *defg(i)*.

push(elem, stack) : empilha **elem** em **stack**.

pop(stack) : retira e retorna o elemento do topo da pilha **stack**.

set_bit(num, conj) : ajusta com o valor "1" o "bit" de número **num** no conjunto de "bits" **conj**.

inter_bit(conj1,conj2) : faz a interseção entre dois conjuntos de "bits" e retorna o conjunto resultado.

neg_bit(conj) : faz a negação dos elementos do conjunto de "bits" **conj** e retorna o novo conjunto obtido.

sub_bit(conj1,conj2) : subtrai o conjunto de "bits" **conj2** de **conj1** retornando o conjunto obtido.

e_vazio_bit(conj) : retorna **true** se todos os "bits" de **conj** forem igual a "0" e **false** caso contrário.

atualiza_pilcam(elem) : atualiza o apontador da pilha **pil_cam** de forma que o topo da pilha seja sempre igual a **elem**.

head(elemento.apont) : retira e retorna um elemento da lista de elementos do tipo LIST apontada por **elemento.apont**.

e_essencial(fonte, alvo, g_suc) : verifica no GFC apontado por **g_suc** se o ramo definido por (**fonte**, **alvo**) é um ramo essencial estendido de fluxo de dados. **e_essencial** retorna **true** se for essencial estendido e **false**, caso contrário.

insere_grafo_i(elemento, nosucg, pdes[i].grafo_i, tipo) : está função insere um novo ramo no grafo(i), criando um novo nó; este ramo tem o nó fonte definido por **elemento.infosuc** e o nó alvo por **nosucg**. **pdes[i].grafo_i** é o ponteiro para o início da lista de nós do grafo(i) e **tipo** indica se o nó alvo (que está sendo inserido no grafo(i)) possui número no GFC idêntico a algum outro nó do grafo(i); nesta situação, o valor **tipo** é REP; caso não exista nó com número igual, o valor de **tipo** é N_REP.

ins_arc(pdes[i].prim_ramos, alvo, fonte, deff) : insere na lista de apontada por **pdes[i].prim_ramos** o ramo definido por (**fonte**, **alvo**) (se ele já não estiver na lista) e associa o conjunto de "bits" **deff** ao ramo (**fonte**, **alvo**).

existe_no_igual(nosucg,pdes[i].grafo_i) : retorna **true** se existir um nó no grafo(i) apontado por **pdes[i].grafo_i** com número no GFC igual a **nosucg.num_no_G** e **deff** igual a **nosucg.deff**; e **false**, caso contrário.

liga_grafo_i(elemento, nosucg, pdes[i].grafo_i) : faz a ligação do nó definido por **elemento.infosuc** para o nó definido pro **nosucg** no grafo(i) apontado por **pdes[i].grafo_i**.

lista_sucessores(x, g_suc) : cria, a partir do GFC, uma lista de sucessores de **x** cujos elementos são do tipo LIST. Os elementos da lista são iniciados com os números dos sucessores no GFC e seus respectivos conjuntos **defg(i)**.

faca_ramo_pfi(elemento, nosucg, pdes[i].grafo_i) : altera o estado do ramo definido por (**elemento.infosuc.num_no_G**, **nosucg.num_no_G**) do grafo(i) apontado por **pdes[i].grafo_i** para p_ϕ.

dom(alvo, fonte) : esta função verifica se o nó **alvo** é dominador do nó **fonte** no GFC, retornando **true** em caso afirmativo e **false**, caso contrário.

```
begin
/* determina os conjuntos de no's do program que possui
   definicao de varia'veis */
conodef = det_conodef(conodef);
i = 1;
while conodef <> nil do
  begin
  x = ret_elem(conodef);
  /* inicia ref com as informacoes do no' x */
  ref.infosuc.num_no_G = x;
  ref.infosuc.def = uniao_bit(info_no[x].defgi,info_no[x].def_ref);
  ref.apont = lista_sucessores(x, g_suc);
  ini_info_descritores(pdes[i]);
  grfi_inicializacao(pdes[i], ref);
  push(ref, conjsuc);
  push(x, pil_cam);
  while conjsuc <> nil do
    begin
    elemento = pop(conjsuc);
    fonte = elemento.infosuc.num_no_G;
    set_bit(fonte, pdes[i].Ni);
    atualiza_pilcam(fonte, pil_cam);
    deff = elemento.infosuc.deff;
    if elemento.apont <> nil then
      nosucg = head(elemento.apont);
    if elemento.apont <> nil then
```

```
      push(elemento, conjsuc);
    alvo = nosucg.num_no_G;
    if e_essencial(fonte,alvo,g_suc) then
      ins_arc(pdes[i].prim_ramos,fonte,alvo,deff);
    deff = sub_bit(
deff,inter_bit(deff,inter_bit(info_no[alvo].defgi,neg_bit(info_no[i].undef))));
    nosucg.deff = deff;
    if !e_vazio_bit(deff) then
      begin
      if  alvo == ref.infosuc.num_no_G then
        begin
        set_bit(alvo,pdes[i].Nt);
        insere_grafo_i(elemento,nosucg,pdes[i].grafo_i,REP);
        faca_ramo_pfi(elemento, nosucg, pdes[i].grafo_i);
        ins_arc(pdes[i].prim_ramos,alvo,fonte,elemento.infosuc.deff);
        end
      else
        if existe_no_igual(nosucg, pdes[i].grafo_i) then
          liga_grafo_i(elemento, nosucg, pdes[i].grafo_i);
           if pertence_pil_cam(alvo,pil_cam) and dom(alvo,fonte)
             faca_ramo_pfi(elemento, nosucg, pdes[i].grafo_i);
        else
          begin
          insere_grafo_i(elemento,nosucg,pdes[i].grafo_i,N_REP);
           if pertence_pil_cam(alvo,pil_cam) and dom(alvo,fonte)
             faca_ramo_pfi(elemento, nosucg, pdes[i].grafo_i);
          push(alvo,pil_cam);
          elem_aux.infosuc = nosucg;
          elem_aux.apont = lista_sucessores(nosucg.num_no_G,g_suc);
          if elem_aux.apont <> nil then
            push(elem_aux,conjsuc);
          else
            begin
            set_bit(alvo,pdes[i].Ni);
            set_bit(alvo,pdes[i].Nt);
            end
          end
```

```
      end
    else  /* deff = vazio */
      begin
      if  alvo == ref.infosuc.num_no_G then
        begin
        set_bit(alvo,pdes[i].Nt);
        insere_grafo_i(elemento,nosucg,pdes[i].grafo_i,REP);
        faca_ramo_pfi(elemento, nosucg, pdes[i].grafo_i);
        ins_arc(pdes[i].prim_ramos,alvo,fonte,elemento.infosuc.deff);
        end
      else
        if existe_no_igual(nosucg, pdes[i].grafo_i) then
          begin
          set_bit(alvo,pdes[i].Nt);
          liga_grafo_i(elemento, nosucg, pdes[i].grafo_i);
          faca_ramo_pfi(elemento, nosucg, pdes[i].grafo_i);
          ins_arc(pdes[i].prim_ramos,alvo,fonte,elemento.infosuc.deff);
          end
        else
          begin
          set_bit(alvo,pdes[i].Nt);
          insere_grafo_i(elemento,nosucg,pdes[i].grafo_i,N_REP);
          faca_ramo_pfi(elemento, nosucg, pdes[i].grafo_i);
          ins_arc(pdes[i].prim_ramos,alvo,fonte,elemento.infosuc.deff);
          end
        end
      end
    end
  i := i + 1;
  end
end
```

E.6.2 Algoritmo para Determinar os Conjuntos D_m

Considere-se que os conjuntos $deff(k_1), \ldots, deff(k_q)$ são representados pelas variáveis $DEFF(k_1)$, $\ldots$, $DEFF(k_q)$ que armazenam conjuntos de variáveis.

1. Determine o conjunto de variáveis comuns a todos $DEFF(k_q)$:

- $D_1 = DEFF(k_1) \bigcap \ldots \bigcap DEFF(k_q)$

2. Subtraia o conjunto D_1 dos conjuntos $DEFF(k_q)$:

   - **for** $I = 1 \ldots q$ **do**
     $DEFF(k_I) = DEFF(k_I) - D_1$;
     done

3. Ordene $DEFF(k_1)$, ..., $DEFF(k_q)$ em ordem crescente de cardinalidade e renomeie os conjuntos como: D_2, ..., D_{q+1}.

4. Torne os conjuntos D_2, ..., D_{q+1} disjuntos:

   - **for** $I = 2 \ldots q+1$ **do**
     for $J = 2 \ldots q+1$ **do**
     if $D_J \neq D_I$ **then**
     $D_J = D_J - D_I$;
     done
     done

5. Elimine conjuntos D_1, ..., D_{q+1} vazios:

   - **for** $I = 1 \ldots q+1$ **do**
     if $D_I = \phi$ **then**
     Elimine D_I;
     done

6. Renomeie os conjuntos restantes: D_1, ..., D_m.

E.7 Considerações Finais

Neste apêndice foram apresentadas as alterações nos algoritmos da POKE-TOOL utilizados na determinação das *adpu*s baseadas em ramos essenciais estendidos. Essas alterações foram realizadas porque, em algumas situações especiais, os conjuntos adequados a essas *adpu*s não eram adequados aos critérios todos os potenciais usos na presença de caminhos não-executáveis. As novas *adpu*s (obtidas a partir dos algoritmos modificados) não estão sujeitas às situações acima e podem ser usadas na determinação de conjuntos adequados aos critérios todos os potenciais usos e todos os potenciais usos/du mesmo na presença de caminhos não-executáveis.

ÍNDICE