Depuração automática de programas baseada em modelos: uma abordagem hierárquica para auxílio ao aprendizado de programação

Wellington Ricardo Pinheiro

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para obtenção do título de
Mestre em Ciências

Programa: Ciência da Computação
Orientadora: Prof.ª Dr.ª Leliane Nunes de Barros

— São Paulo, maio de 2010 —

Depuração automática de programas baseada em modelos: uma abordagem hierárquica para auxílio ao aprendizado de programação

Este exemplar corresponde à redação final da dissertação, devidamente corrigida e defendida por *Wellington Ricardo Pinheiro*, aprovada pela comissão julgadora.

São Paulo, maio de 2010.

Banca examinadora:

- Prof.ª Dr.ª Leliane Nunes de Barros (presidente)
  Instituto de Matemática e Estatística (IME-USP)
- Prof. Dr. Evandro de Barros Costa
  Universidade Federal de Alagoas (UFAL)
- Prof. Dr. Nizam Omar
  Universidade Presbiteriana Mackenzie (UPM)

Ao meu tio Antônio Evaristo dos Santos, que sempre serviu de inspiração e, infelizmente, não pôde ver a conclusão desse trabalho.

Agradecimentos

Inicialmente, gostaria de agradecer a todas as pessoas que me apoiaram no decorrer desse mestrado. Sem essas pessoas não teria sido possível concluir mais essa etapa na minha vida. Deixo aqui o meu sincero obrigado.

À professora *Leliane Nunes de Barros*, que sempre acreditou no meu trabalho e me incentivou a continuar mesmo nos momentos mais difíceis. A sua paciência para discutir inúmeros detalhes e as suas valiosas idéias foram imprescindíveis durante o decorrer desse trabalho.

Aos professores *Silvio do Lago Pereira* e *Marco Aurélio Gerosa*, participantes da minha banca de qaulificação; *Evandro de Barros Costa* e *Nizam Omar*, integrantes da banca de defesa, pelos seus comentários e sugestões de melhoria desse trabalho.

Aos professores do Departamento de Ciência da Computação do IME-USP, que contribuíram para a minha formação nesse mestrado. Em especial, aos professores: *Fabio Kon*, que sempre se interessou pelo meu trabalho e ofereceu a minha primeira bolsa de estudo no IME-USP; *Carlos Hitoshi Morimoto*, que possibilitou que eu conseguisse uma bolsa do projeto Tidia-Ae da FAPESP; e Alfredo Goldman, que, mesmo ausente, permitiu que eu ficasse trabalhando em sua sala.

Aos colegas do laboratório 127A, que sempre me fizeram companhia e proporcionaram boas risadas. Em especial, à *Karina Valdivia Delgado*, cujo trabalho foi precursor desse meu trabalho, e me ajudou em diversos momentos em vários assuntos pesquisados nesse mestrado.

Aos colegas da S4B Digital, que se desdobraram para cobrir a minha ausência no serviço, em inúmeras situações. Em especial, ao *Alcides*, amigo e chefe, que permitiu que eu continuasse a minha pesquisa mesmo tendo que sacrificar várias horas de serviço.

À minha esposa *Sandra*, que suportou desde a data do nosso casamento os meus diversos momentos de ausência, às limitações de tempo e o meu mal humor. Ainda assim, sempre com muito amor e carinho me motivou a seguir em frente.

À minha mãe, *Maria Inês dos Santos Pinheiro*, ao meu pai, *José Benedicto Pinheiro*, e meus irmãos que, apesar da minha ausência no decorrer desse trabalho, sempre me incentivaram.

A todos os meu amigos, que tiveram a paciência de esperar pelo término do mestrado.

Por fim, agradeço a Deus, que permitiu que eu concluísse esse trabalho e também por ter colocado na minha vida as pessoas maravilhosas que encontrei nesse caminho que optei por seguir.

Resumo

Diagnóstico baseado em modelos (*Model Based Diagnosis* - MBD) é uma técnica de Inteligência Artificial usada para encontrar componentes falhos em dispositivos físicos. MBD também tem sido utilizado para auxiliar programadores experientes a encontrarem falhas em seus programas, sendo essa técnica chamada de Depuração de Programas baseada em Modelos (*Model Based Software Debugging* - MBSD). Embora o MBSD possa auxiliar programadores experientes a entenderem e corrigirem suas falhas, essa abordagem precisa ser aprimorada para ser usada por aprendizes de programação. Esse trabalho propõe o uso da técnica de depuração hierárquica de programas, uma extensão da técnica MBSD, para que aprendizes de programação sejam capazes de depurar seus programas raciocinando sobre componentes abstratos, tais como: padrões elementares, funções e procedimentos. O depurador hierárquico de programas proposto foi integrado ao Dr. Java e avaliado com um grupo de alunos de uma disciplina de Introdução à Programação. Os resultados mostram que a maioria dos alunos foi capaz de compreender as hipóteses de falha geradas pelo depurador automático e usar essas informações para corrigirem seus programas.

Abstract

Model Based Diagnosis (MBD) in Artificial Intelligence is a technique that has been used to detect faulty components in physical devices. MBD has also been used to help senior programmers to locate faults in software with a technique known as Model Based Software Debugging (MBSD). Although this approach can help experienced programmers to detect and correct faults in their programs, this approach must be improved to be used with novice programmers. This work proposes a hierarchical program diagnosis, a MBSD extension, to help novice programmers to debug programs by exploring the idea of abstract components, such as: elementary patterns, functions and procedures. The hierarchical program debugger proposed was integrated to the Dr. Java tool and evaluated with students of an introductory programming course. The results showed that most of the students were able to understand the hypotheses of failure presented by the automated debugger and use this information to provide a correction for their programs.

Sumário

Lista de Siglas **v**

1 Introdução **1**

1.1 Motivação . . . 3

1.2 Objetivos . . . 7

1.3 Organização . . . 7

I Fundamentos **9**

2 Aprendizagem de Programação: Conceitos Básicos **11**

2.1 Falhas típicas de programação . . . 11

2.1.1 Taxionomia de falhas em programas . . . 12

2.2 Ensino/Aprendizagem de programação auxiliada por padrões . . . 19

2.3 Depuração de programas . . . 20

2.4 Trabalhos correlatos . . . 22

2.4.1 PROUST . . . 23

2.4.2 ProPAT . . . 25

2.5 Considerações finais . . . 26

3 Diagnóstico Baseado em Modelos **27**

3.1 Conceitos básicos de Diagnóstico Baseado em Modelos . . . 27

3.2 Modelos centrados em componentes . . . 29

3.3 Tarefas do processo de diagnóstico . . . 30

3.4 Formalização do problema de diagnóstico . . . 30

3.5 Algoritmo de Reiter: *Minimal Hitting Set* 34
3.6 MBD com o algoritmo de Reiter 38
3.7 Exemplo de diagnóstico com MBD 43
3.7.1 Exemplo de Diagnóstico com o Algoritmo de Reiter 43
3.7.2 Discriminação de Hipóteses 43
3.8 Complexidade do MBD 45
3.9 Considerações finais 45

4 Diagnóstico Hierárquico Baseado em Modelos **47**

4.1 Conceitos básicos 48
4.1.1 Modelo Abstrato 48
4.1.2 Modelos com Múltiplos níveis de abstração 49
4.1.3 Representação de componentes abstratos 49
4.2 Solução para um problema de HMBD 53
4.3 Abordagem *top-down* para fazer o diagnóstico hierárquico 53
4.4 Algoritmo para fazer o diagnóstico hierárquico 56
4.5 Exemplo de HMBD 56
4.6 Desempenho do HMBD 57
4.7 Considerações finais 58

5 Depuração de Programas Baseada em Modelos **61**

5.1 Diagnóstico *versus* depuração 61
5.2 Depuração de programas baseada em modelos 62
5.3 Modelo Baseado em Valor 63
5.3.1 Expressões 64
5.3.2 Atribuições 66
5.3.3 Seleções 66
5.3.4 Repetições 67
5.3.5 Comando return 67
5.3.6 Chamada a métodos que não devolvem valores 67

5.3.7 Chamada a métodos que devolvem valores 68
5.4 Transformação do programa em um modelo baseado em valor 68
5.5 Exemplo de diagnóstico com MBSD 69
5.5.1 Detecção de sintomas 69
5.5.2 Construção do modelo baseado em valor 70
5.5.3 Geração de hipóteses 72
5.6 Considerações finais 74

II Dr. Java Pro **75**

6 Depuração Hierárquica de Programas **77**
6.1 Limitações do MBSD no processo de aprendizagem de programação 77
6.2 Depuração de programas com abstrações 78
6.3 Representação de abstrações programas 79
6.4 Discriminação de Hipóteses 82
6.5 O algoritmo HPD 85
6.6 Exemplo de depuração com o HPD 86
6.7 Considerações finais 88

7 Implementação do Algoritmo HPD **91**
7.1 Implementação do depurador automático de programas 91
7.1.1 Análise sintática 92
7.1.2 Construção do modelo 93
7.1.3 Transformação de entradas/saídas de um caso de teste em observações 95
7.1.4 Propagação de restrições 96
7.1.5 Geração das hipóteses de falha 98
7.1.6 Discriminação de hipóteses 99
7.2 Dr. Java Pro: uma extensão do Dr. Java com depurador automático 102
7.3 Considerações finais 111

III Avaliação Experimental e Conclusões **113**

8 Verificação e avaliação da Ferramenta Dr. Java Pro **115**

8.1 Verificando a implementação do Dr. Java Pro . . . 115

8.2 Avaliação do uso do Dr. Java Pro por um grupo de alunos . . . 117

8.3 Avaliação do Dr. Java Pro realizada pelos alunos . . . 126

8.4 Conclusão das Análises Experimentais . . . 128

9 Conclusão e Trabalhos Futuros **131**

9.1 Principais contribuições . . . 132

9.2 Trabalhos futuros . . . 133

9.2.1 Implementação . . . 133

9.2.2 Técnicas de diagnóstico e depuração automática . . . 133

9.2.3 Avaliação pedagógica . . . 134

IV Apêndices **137**

A Padrões elementares **139**

B Problemas de teste **143**

C Arquivo de configurações de problemas de testes **147**

D Questionário de Avaliação da Ferramenta **149**

Lista de Siglas

MBD	Diagnóstico baseado em modelos (*Model Based Diagnosis*).
HMBD	Diagnóstico hierárquico baseado em modelos (*Hierarchical Model Based Diagnosis*).
MBSD	Depuração de programas baseada em modelos (*Model Based Software Debugging*).
HPD	Diagnóstico hierárquico de programas (*Hierarchical Program Debugging*).
VBM	Modelo baseado em valor (*Value Based Model*).
Dr. Java Pro	Extensão do Dr. Java com o algoritmo HPD proposto nessa dissertação.
Dr. Java	Ambiente para desenvolvimento de programas para aprendizes de programação em Java.
PROUST	Uma importante ferramenta de aprendizado de programação, muito citado na literatura.
PROPAT	Depurador automático de programas que precedeu o depurador HPD, proposto nesse trabalho.

Capítulo 1

Introdução

Aprender a programar é uma tarefa difícil pois envolve: (i) o aprendizado de uma linguagem de programação e (ii) como resolver problemas escrevendo programas nessa linguagem. Pesquisas apontam que mesmo para um aluno que já possui conhecimento a respeito da sintaxe e semântica de uma linguagem de programação, aprender como combinar as sentenças dessa linguagem de maneira a construir um programa que resolva um determinado problema é uma tarefa muito difícil [47]. Isto é, escrever a solução para um problema (por exemplo, resolver equações de 2o grau), numa linguagem de programação (conhecida ou não), ainda é a principal dificuldade enfrentada por um aprendiz de programação.

Numa ferramenta de suporte ao aprendizado de programação, a grande dificuldade é como avaliar o programa feito por um aluno para resolver um problema de programação, isto é, como depurar programas de forma automática. Técnicas de depuração automática requerem a especificação formal do programa do aluno por um especialista e não são voltadas para interação com o aluno [16]. PROUST [25] é uma ferramenta de ensino de programação bem conhecido, capaz de realizar a depuraração automática de programas. Com base no programa do aluno, eventualmente incorreto, PROUST tenta classificar fragmentos do programa como *planos de programação* previamente armazenados em uma biblioteca de planos e metas (intenções). Essa abordagem apresenta duas desvantagens: (1) necessidade de um especialista para fornecer os planos armazenados na biblioteca e; (2) não há garantia de que todas as possíveis soluções para um problema serão cobertas pelos planos armazenados na biblioteca.

Em [3,15] foi proposto um sistema que utiliza técnicas de **Diagnóstico baseado em Modelos** (*Model Based Diagnosis* - MBD) [36,14], capaz de encontrar falhas no programa do aluno. A vantagem que essa abordagem tem, com relação a adotada pelo PROUST [25], é que não há necessidade de usar uma biblioteca de planos e metas previamente especificada para um conjunto de problemas.

Diagnóstico é a tarefa de identificar os componentes falhos em um sistema cujo comportamento pode ser previsto. De um lado, está o *sistema* S (composto por um conjunto de componentes) cujo comportamento, possivelmente falho, pode ser observado. Do outro lado, está um *modelo do sis-*

tema, M_S, que descreve o comportamento correto de S numa linguagem formal e é usado para fazer predições a respeito das entradas e saídas dos componentes. O modelo M_S, pode descrever, através de um conjunto de axiomas, quais são os componentes de S, como eles estão conectados e o comportamento de cada um. A diferença entre uma observação e uma predição é chamada de *discrepância* (ou erro), enquanto uma equivalência é chamada de *concordância*. As discrepâncias e as concordâncias são usadas para identificar os componentes falhos no sistema [5]. Uma variação do MBD é o *Diagnóstico Hierárquico baseado em Modelo* (*Hierarchical Model Based Diagnosis* - HMBD) [31,8]. Nesse caso, o modelo S é descrito de forma hierárquica através de componentes abstratos que, durante o processo de diagnóstico, podem ser sucessivamente decompostos até resultar nos componentes do modelo M_S que correspondem aos componentes reais de S.

Apesar de ser normalmente utilizada para detectar falhas em dispositivos físicos, a técnica MBD também tem sido usada na tarefa de depuração de programas [9,26,27,46,28]. Enquanto um engenheiro verifica os sistemas mecânicos ou elétricos para localizar os componentes falhos, tentando compreender a diferença entre o comportamento apresentado e seus modelos corretos, um professor de programação tenta entender a diferença entre o programa do aluno e suas *intenções* (uma vez que existem inúmeras maneiras de resolver um problema de programação). Sendo o programa do aluno visto como um sistema no qual deve ser feito o diagnóstico, é interessante observar que o modelo correto do sistema para fazer as predições não está disponível, mas somente o programa do aluno, contendo falhas. Com o uso de modelos corretos que descrevem a semântica da linguagem de programação e de casos de teste que, de uma certa maneira declaram as *intenções* que qualquer programador deve ter ao elaborar seu programa com relação às entradas e saídas, é possível fazer hipóteses sobre as possíveis falhas no programa.

O uso de diagnóstico baseado em modelos para depuração de programas, também chamado de **Depuração de Programas baseada em Modelos** (*Model Based Software Debugging* -**MBSD**), foi originalmente proposto em Console et. al (1993) e posteriormente melhor explorado em Mateis et. al. (2000), com a proposta de ajudar programadores experientes a encontrarem falhas em seus programas. No modelo do programa, as expressões e sentenças da linguagem são representadas como componentes (sem hierarquia, em geral), enquanto a conexão entre componentes representam um fluxo de informação entre sentenças ou expressões. O comportamento dos componentes é derivado da semântica da linguagem. Os valores observados são os processados pelo programa e os desejados são definidos: (i) pelos casos de teste ou (ii) pelo programador em pontos específicos do programa. O diagnóstico obtido é um conjunto de sentenças e expressões do programa que podem estar falhas, isto é, escritas de maneira incorreta.

Um aspecto interessante na depuração automática de programas de alunos é que, além de fornecer informações para construir o modelo do aluno, o próprio processo de depuração pode promover o aprendizado: um efeito colateral que pode ocorrer em todas as metodologias de ensino/aprendizagem centradas na resolução de problemas. Assim, o interesse desse trabalho é explorar as possibilidades de aprendizagem durante o processo de depuração automática do programa

do aluno usando a técnica de MBSD.

1.1 Motivação

Exemplo 1.1 - Problema da ordem crescente. *Dados dois números inteiros, a e b, construir um programa que imprima esses números em ordem crescente. Os casos de teste que deverão ser usados para testar o seu programa solução são*:

Caso de teste	**Entradas**	**Saídas**
Teste 1	3, 5	3, 5
Teste 2	5, 3	3, 5
Teste 3	3, 3	3, 3

□

Programa 1 Programa construído por um aluno para solucionar o problema da ordem crescente, gerando saídas incorretas.

```
1  public class OrdemCrescente {
2    public static void crescente(int a, int b) {
3      int maior;
4      int menor;
5
6      if (a < b) {
7        maior = a;
8        menor = b;
9      } else {
10       maior = b;
11       menor = a;
12     }
13
14     writeInt("o menor numero e'", menor);
15     writeInt("o maior numero e'", maior);
16   }
17 }
```

O Programa 1, escrito na linguagem Java, mostra a solução proposta por um aluno para o problema do Exemplo 1.1 e não apresenta erros de sintaxe ao ser compilado. No entanto, o programa só passa com sucesso no Teste 3, isto é, o programa apresentou um erro na saída para os Testes 1 e 2. Suponha que o aluno tenha implementado corretamente a sua estratégia de solução porém, ele introduziu uma falha na linha 6 do programa, invertendo o operador na expressão condicional do comando de seleção alternativa, isto é, ao invés de usar “>” ele usou o operador “<”. Com isso, as saídas foram invertidas, causando erros nos Testes 1 e 2. A falha no programa cometida pelo aluno pode ter origem em seu conhecimento incorreto sobre a avaliação de uma expressão lógica

(isso pode ocorrer com mais frequência na avaliação de expressões lógicas mais complexas que a do Exemplo 1.1). Chamaremos esse erro conceitual do aluno de ECA-1. Uma outra possibilidade é que o aluno tenha intencionalmente usado o operador "<" na Linha 6, mas inverteu os blocos de comandos da seleção alternativa (Apêndice A), isto é, o aluno trocou as Linhas 7 e 8 pelas Linhas 10 e 11 (para esse exemplo, isso é o mesmo que inverter os valores das duas atribuições em cada bloco). Esse tipo de falha pode ter origem num conhecimento incorreto do aluno sobre a semântica do comando *if-then-else*. Chamaremos esse erro conceitual do aluno de ECA-2.

Ao chamarmos um depurador automático de programas baseado em modelos (MBSD) para o Programa 1, isto é, que implementa a técnica de MBSD mencionada anteriormente, ele devolveria ao aluno um conjunto de todas as hipóteses de falha que *explicam corretamente* o comportamento errado do programa do aluno para o Teste 1, a saber:

$$\{\{6\}, \{7, 8\}, \{14, 15\}, \{8, 15\}, \{7, 14\}\}$$

Cada uma dessas hipóteses de falha contém uma ou mais linhas do programa que podem ser consideradas como as falhas do programa. Por exemplo: a hipótese de falha única $\{6\}$ significa que somente alterando a Linha 6 do programa, é possível obter um programa correto para o Teste 1; a hipótese de falha múltipla $\{7, 8\}$ significa que é possível corrigir o programa modificando-se ambas as linhas 7 e 8. Uma característica importante da técnica de Depuração de Programas baseada em Modelos é ser possível corrigir um programa falho, corrigindo *qualquer uma das hipóteses de falha* geradas pelo depurador automático [1]. Dessa forma, podemos considerar que, para um determinado caso de teste, qualquer uma das hipóteses pode ser utilizada para corrigir o programa falho do aluno (note que as Linhas 10 e 11 não aparecem no conjunto de hipóteses de falha. Isso ocorre porque, para o Teste 1, essas linhas não são executadas).

Hipótese	**Correção**
$\{6\}$	trocar o operador ">" pelo operador "<" na Linha 6
$\{7, 8\}$	inverter os valores nas atribuições das Linhas 7 e 8
$\{14, 15\}$	inverter somente os nomes das variáveis impressas nas Linhas 14 e 15
$\{8, 15\}$	na Linha 8 mudar para $menor = a$ e na Linha 15, imprimir o valor de b
$\{7, 14\}$	na Linha 7 mudar para $maior = b$ e na Linha 14, imprimir o valor de a

Tabela 1.1: Correções para as hipóteses devolvidas pelo depurador automático. Todas as correções indicadas são capazes de corrigir o programa para o Teste 1.

A Tabela 1.1 mostra as correções que podem ser feitas (pelo programador ou aluno) para as hipóteses de falha geradas pelo depurador automático MBSD, com o caso de Teste 1. Observe que as correções na Tabela 1.1 das hipóteses de falha $\{7, 8\}$, $\{8, 15\}$ e $\{7, 14\}$, tornam o programa correto,

[1] No diagnóstico baseado em modelos de sistemas físicos, em geral, somente uma das hipóteses de falha é a verdadeira. Isto é, somente uma possível combinação de componentes físicos estão falhos e devem ser reparados para corrigir o comportamento falho apresentado pelo sistema.

mas somente para o Teste 1. Por outro lado, as hipóteses $\{6\}$ e $\{14, 15\}$, com suas correspondentes correções, corrigem as falhas no programa para o Teste 1 e para todos os demais testes (para outros exemplos e programas com falhas, podem existir situações em que não exista uma correção para todos os casos de teste que apresentarem erro na saída).

A hipótese $\{6\}$ está relacionada ao erro conceitual do aluno ECA-1. Assim, supondo que essa tenha sido a origem da falha cometida, espera-se que, ao apresentar essa hipótese ao aluno, esse seja capaz de fazer a correção correspondente da Tabela 1.1.

A hipótese $\{7, 8\}$ está, de alguma forma, relacionada ao erro conceitual do aluno ECA-2 (inversão dos blocos de comandos da seleção alternativa). Porém, como o Teste 1 só executou o bloco do *then*, o sistema de depuração apontou que uma alteração somente nas Linhas 7 e 8 pode corrigir o programa. Assim, a correção indicada na tabela corresponde a fazer uma cópia do bloco *else* nas Linhas 7 e 8. Observe que essa alteração não faz com que o programa passe no Teste 2, o que causará uma segunda chamada ao depurador automático.

A hipótese de falha $\{14, 15\}$ – *inverter somente os nomes das variáveis impressas nas Linhas 14 e 15* – corrige o programa para todos os casos de teste porém, o programa torna-se ilegível pois usa as variáveis mnemônicas, *menor* e *maior*, de maneira incorreta, sendo essa uma correção não intuitiva para um aluno iniciante fazer.

As hipóteses de falha $\{8, 15\}$ e $\{7, 14\}$ (uma combinação das hipóteses anteriores $\{7, 8\}$ e $\{14, 15\}$), dificilmente estarão relacionadas diretamente a uma falha cometida por um aluno iniciante, sendo que este encontraria dificuldades em corrigi-las. No entanto, fazer com que o aluno encontre uma correção alterando somente as linhas indicadas pelo depurador, pode ser considerado um exercício de programação interessante. Note que, tanto a correção proposta na Tabela 1.1 para a hipótese $\{8, 15\}$ como para a $\{7, 14\}$, faz com que o programa obtenha sucesso para o Teste 1. Essa é uma característica importante de um depurador automático: por ser um sistema baseado num formalismo lógico, isto é, que usa uma teoria lógica para descrever o comportamento de programas, ele é capaz de gerar *somente hipóteses corretas de falhas de programas*, isto é, falhas para as quais existam correções que tornam o programa correto para um dado caso de teste.

Em suma, podemos usar o conjunto de hipóteses de falha geradas por um depurador automático baseado em modelos, como exercícios de programação para que o aluno encontre diferentes formas de corrigir o seu programa, ou então: (i) *filtrar* o conjunto de hipóteses de falha antes de apresentá-las ao aluno, eliminando assim as falhas menos plausíveis (como por exemplo, comandos de impressão e leitura); (ii) *discriminar as hipóteses*, uma a uma, através de interações com aluno, a fim de identificar erros conceituais do aluno que resultaram na falha cometida no programa [2].

[2]A técnica de discriminação de hipóteses num processo de diagnóstico de sistemas físicos, consiste em realizar testes e medições nas entradas e saídas de componentes, coletando assim novas informações que diminuam o conjunto de hipóteses. Um paralelo na depuração de software seria perguntar ao programador/aluno os valores de variáveis esperados por ele em diferentes pontos do seu programa.

Vantagens e desvantagens do uso de um depurador automático de programas no auxílio do aprendizado de programação. Através do Exemplo 1.1, mostramos de maneira informal que um depurador automático baseado em modelos pode ser usado como uma valiosa ferramenta para auxílio ao aprendizado de programação, dado que ele é capaz de:

- gerar um conjunto completo de hipóteses (corretas) de falha sobre o programa do aluno quando executado para um caso de teste e;
- questionar o aluno sobre diferentes maneiras de corrigir um programa com falhas, promovendo assim o aprendizado.

Apesar dessa técnica poder auxiliar um aluno no aprendizado de programação, ela apresenta as seguintes limitações:

1. mesmo para um programa pequeno, o sistema de diagnóstico pode apontar diversas sentenças e expressões do programa como hipóteses de falha. Apresentar um conjunto grande de hipóteses de falha ao aluno pode confundi-lo, ao invés de ajudá-lo, na correção do programa;
2. apenas mostrar as hipóteses de falha através das linhas do programa pode trazer pouca informação para que um aluno iniciante consiga corrigir o seu programa;
3. considerando a etapa de discriminação de hipóteses, um aluno iniciante nem sempre sabe expressar suas intenções ou fornecer informações sobre o comportamento de seus próprios programas (i.e., em termos de valores de variáveis em pontos de seu programa).

Para tratar essas três limitações, propomos o uso de um depurador automático baseado em modelos hierárquicos, isto é, que considere componentes abstratos em suas hipóteses de falha. Por exemplo, no Programa 1, podemos chamar de componente abstrato, o comando de seleção alternativa da Linha 6 até a Linha 12. Assim, um *depurador automático hierárquico* devolveria o seguinte conjunto de hipóteses de falha para o Programa 1:

$$\{\{SelecaoAlternativa[6-12]\},\{14,15\}\}$$

o que faria com que o aluno analisasse um número menor de hipóteses. No nosso exemplo, caso o aluno opte por depurar as linhas do componente abstrato $SelecaoAlternativa[6-12]$, seria feita uma nova chamada ao depurador automático e as demais hipóteses (nesse caso, {14,15}) seriam descartadas. Outros tipos de componentes abstratos podem ser identificados no programa do aluno na forma de funções, procedimentos e *padrões elementares de programação* [6, 1, 17, 44] (por exemplo, os padrões *repetição contada*, *repetição com sentinela* ou *repetição com indicador de passagem*, inicialização de vetor ou matriz, etc). Acreditamos que o aluno com algum conhecimento

sobre a sintaxe e semântica de funções, procedimentos e padrões elementares[3], terá mais informação para decidir se a falha cometida no programa está realmente nesse componente e encontrar a melhor correção de seu programa.

1.2 Objetivos

Esse trabalho tem como objetivo principal desenvolver um sistema de depuração automática baseado em modelos hierárquicos, que chamaremos de **Depurador Hierárquico de Programas** (*Hierarchical Program Debugging* - HPD), capaz de gerar hipóteses de falha sobre componentes abstratos e detectar falhas lógicas cometidas por um aluno iniciante de programação, em programas sem erros de compilação. Tais componentes abstratos podem ser identificados no programa do aluno na forma de funções, procedimentos e *padrões elementares de programação* [6, 1, 17, 44]. Com isso, esperamos que o uso do depurador automático HPD possa ser usado como uma ferramenta de auxílio no aprendizado de programação, de forma mais eficaz do que a usada nas técnicas tradicionais (i.e., não hierárquicas) de depuração de programas baseada em modelos (MBSD) [26, 27, 30, 29, 15]. Mais especificamente, esse trabalho se propõe a:

1. Apresentar os conceitos de Diagnóstico baseado em Modelos (MBD) e Diagnóstico Hierárquico baseado em Modelos (HMBD).
2. Mostrar como a técnica de MBD pode ser usada para fazer a depuração de programas, chamada de *Depuração de Programas baseada em Modelos* (MBSD).
3. Integrar as técnicas de MBSD e HMBD para fazer a depuração de programas utilizando abstrações (padrões elementares, funções e procedimentos), resultando no sistema proposto (HPD).
4. Desenvolver o sistema HPD e integrá-lo à plataforma de programação Dr. Java. Chamamos essa integração de **Dr. Java Pro** – Dr. Java Professor.
5. Realizar um experimento e apresentar os resultados obtidos da avaliação da ferramenta realizada com um grupo de alunos de uma disciplina de Introdução à Programação.

1.3 Organização

Essa dissertação está organizada da seguinte forma:

[3] Espera-se que além de funções e procedimentos, padrões elementares de programação também sejam apresentados ao aluno em sala de aula ou na forma de apostilas.

Capítulo 2	Apresentamos os conceitos básicos relacionados à aprendizagem de programação: tipos de falhas em programas; aprendizagem baseada em padrões elementares de programação; processos de depuração de programas e ferramentas de depuração; e trabalhos correlatos.
Capítulo 3	Introduzimos os conceitos de Diagnóstico baseado em Modelos e de um importante algoritmo utilizado para fazer o diagnóstico chamado *Minimal Hitting Set*.
Capítulo 4	Introduzimos os conceitos do Diagnóstico Hierárquico baseado em Modelos, uma importante extensão do Diagnóstico baseado em Modelos, que permite que o diagnóstico seja feito em diversos níveis de abstração.
Capítulo 5	Apresentamos a técnica de Depuração de Programas baseada em Modelos, que é uma aplicação do Diagnóstico baseado em Modelos para fazer a depuração de programas.
Capítulo 6	Apresentamos uma nova abordagem para fazer depuração de programas, chamada de Depuração Hierárquica de Programas, que une as técnicas de MBSD e HMBD, e permite depurar os programas em diversos níveis de abstração.
Capítulo 7	Mostramos como foi feita a implementação do Dr. Java Pro e do algoritmo de Depuração Hierárquica de Programas.
Capítulo 8	Apresentamos os resultados da avaliação do nosso sistema e também como foi feita a verificação do nosso código implementado.
Capítulo 9	Concluímos com as principais contribuições desse trabalho e os possíveis trabalhos futuros.
Apêndice A	Descrevemos os padrões elementares citados nesse trabalho.
Apêndice B	Mostramos uma descrição curta dos problemas de teste que usamos para verificar a implementação do sistema HPD.
Apêndice C	Apresentamos como é feita a configuração dos problemas de testes, para serem executados de forma automatizada.
Apêndice D	Apresentamos o questionário usado pelos alunos que participaram dos experimentos para avaliarem a ferramenta construída.

Os Capítulos 2, 3, 4 e 5 pertencem à parte de **Fundamentos**, onde discutimos as técnicas e teorias que serviram de base para esse trabalho. As nossas principais contribuições estão na parte **Ambiente Dr. Java Pro**, que compreende os Capítulos 6 e 7. A parte de **Avaliação Experimental e Conclusões** compreende os Capítulos 8 e 9. A parte final desse trabalho contém todos os apêndices.

Parte I

Fundamentos

Capítulo 2

Aprendizagem de Programação: Conceitos Básicos

Nesse capítulo, apresentamos alguns conceitos básicos relacionados ao processo de aprendizagem de programação. Inicialmente, são discutidos os principais tipos de falhas que podem ocorrer em programas de alunos iniciantes, organizadas em uma taxionomia de falhas. Em seguida, discutimos o uso de padrões elementares de programação e técnicas de depuração de programas no ensino/aprendizagem de programação. Finalmente, apresentamos os trabalhos correlatos.

2.1 Falhas típicas de programação

O termo *falha*, conforme utilizado para programas de computadores, pode ser definido como sendo *um passo errado na execução do programa que o impede de funcionar corretamente ou que gera resultados errados na saída* [46]. Apesar de uma falha normalmente surgir na construção de um programa, ela também pode surgir na estratégia utilizada para resolver um problema, ou seja, na especificação dos algoritmos ou estruturas de dados utilizados.

Existem diversos outros termos que são frequentemente utilizados como sinônimos, mas que devem ser diferenciados nesse trabalho conforme a terminologia padrão da IEEE para a engenharia de software [24], a saber:

- *falha* ou *bug* é um passo ou uma definição incorreta de dados. A falha geralmente causa a geração de valores incorretos na saída do programa ou erros de compilação;
- *defeito* é uma consequência da introdução da falha, isto é, um programa que apresenta defeito contém uma ou mais falhas;
- *erro* é a diferença entre um valor computado (observado ou medido) e o valor tido teoricamente como correto.

Além dos termos definidos acima, nessa dissertação, usamos o termo *programa falho* para nos referir a um programa que contém uma ou mais falhas. Usaremos o termo *erro conceitual de programação* (do aluno) para nos referir a um erro no conhecimento do aluno sobre a semântica ou

sintaxe de uma linguagem de programação ou funcionamento do computador, que resulta em uma falha no programa do aluno.

Nas seções a seguir, será apresentada uma taxionomia geral de falhas de programação e as principais falhas encontradas em programas escritos por aprendizes.

2.1.1 Taxionomia de falhas em programas

Conforme apresentado em [46, 15], as falhas em programas podem ser classificadas de acordo com as suas *consequências*, a saber:

- **(C.1)** *falhas em tempo de compilação*, são falhas detectadas pela geração de mensagens de erro ou advertência emitidas pelo compilador;
- **(C.2)** *falhas em tempo de execução*, ocorrem durante a execução do programa, interrompendo-o de forma anormal ou fazendo com que o programa entre em um laço infinito (o programa não para). Esse tipo de falha é geralmente difícil de ser detectado, pois ou o programa é finalizado e não sobra nenhuma informação em memória que permita o rastreamento da falha, ou o programa nunca termina;
- **(C.3)** *falhas que geram erro na saída do programa*, são falhas que terminam a execução do programa de forma normal, mas gera valores incorretos. **Esse tipo de falha é a mais comum de ser detectada por um sistema de depuração automática**;
- **(C.4)** *falhas que não geram erros na saída do programa*, são falhas que ocorrem em programas que apresentam o comportamento esperado para certas entradas, mas ainda assim podem conter falhas (falhas que compensam outras). Esse tipo de falha também é considerada difícil de ser detectado.

As falhas também podem ser classificadas de acordo com a forma como elas se manifestam, a saber:

- **(M.1)** *falhas sintáticas*, são falhas encontradas durante a compilação do programa e indicam que o programa viola alguma regra definida na gramática da linguagem;
- **(M.2)** *falhas semânticas*, violam a semântica definida para a linguagem, tais como: operações com tipos de dados incompatíveis, uso de uma variável não inicializada;
- **(M.3)** *falhas lógicas*, falhas resultantes de erros lógicos do programador, tais como: erro no projeto dos algoritmos, uso de estrutura de dados inadequadas e uso de variáveis erradas. **As falhas lógicas são as principais falhas tratadas por depuradores automáticos de programas**. Esse tipo de falha pode ainda ser dividida em:

- **(M.3.1)** *falhas lógicas funcionais* ou *falhas funcionais*, que resultam no armazenamento de um valor incorreto em alguma variável, para algum caso de teste do programa. Em particular, esse tipo de falha inclui o uso de operadores ou literais errados;
- **(M.3.2)** *falhas lógicas estruturais*, são falhas referentes à estruturação do programa, tais como: uso de variáveis erradas, ausência de sentenças e inversão na ordem de sentenças.

Como mencionamos na introdução, a maior dificuldade que alunos iniciantes enfrentam é como resolver problemas em uma linguagem de programação, sendo que aprender a sintaxe da linguagem é considerado um desafio menor. Assim, durante a construção de um programa, as falhas cometidas por aprendizes de programação ocorrem, normalmente, devido aos conceitos errados a respeito de certas estruturas da linguagem. Essas falhas estão em geral relacionadas às falhas lógicas (M.3). Por exemplo: um aluno acredita que uma variável pode ter mais de um valor ao mesmo tempo ou, em um comando de seleção alternativa (*if-then-else*) o aluno imagina que ambas as partes serão executadas.

Como o modelo que utilizaremos no processo de depuração automática, chamado de *modelo baseado em valor*, permite reconhecer, principalmente, falhas lógicas funcionais (M.3.1)[1] (detalhado no Capítulo 5) e aquelas que geram erro na saída do programa (C.3), nesse trabalho faremos as seguintes suposições sobre falhas em programas de alunos:

- o programa do aluno é compilado corretamente, ou seja, o aluno conhece a sintaxe da linguagem e uma falha do tipo (C.1) ou (M.1) não deverá ocorrer;
- o programa do aluno poderá conter falhas que geram erro na saída do programa (C.3) e, através do uso de casos de testes, será possível detectar programas com falhas e chamar um depurador automático para devolver as hipóteses de falha;
- falhas do tipo (C.2) não poderão ocorrer no programa do aluno uma vez que: se o programa tiver um laço infinito o modelo do programa não poderá ser construído (vide Capítulo 5) e se o programa for interrompido abruptamente, o sistema de depuração proposto também não poderá propagar valores corretamente no modelo do programa;
- falhas que não geram erros na saída do programa (C.4), poderão ocorrer para um ou mais casos de teste, desde que existam outros casos de teste para o programa que recaia no tipo de falha (C.3);
- falhas semânticas (M.2) que não geram erros de compilação poderão ocorrer;
- falhas lógicas funcionais (M.3.1) podem ser detectadas pelo depurador automático baseado em modelos bem como alguns tipos de falhas lógicas estruturais (M.3.2).

[1] O uso desse tipo de modelo num sistema MBSD não é recomendado para encontrar falhas lógicas estruturais (M.3.2) [27], apesar de ser capaz de detectar algumas falhas específicas envolvendo troca de linhas.

A Tabela 2.1 apresenta diversos exemplos de erros conceituais lógicos (M.3), incluindo uma descrição e alguns exemplos.

Descrição	**Exemplo**
O aluno interpreta a atribuição na ordem inversa.	O aluno interpreta que na sentença $a = b$ o valor de a é atribuído a b.
O aluno interpreta uma atribuição como uma comparação matemática.	$x = x + 1$ não pode ser feito pois matematicamente é uma expressão sem resposta.
O aluno acredita que após uma atribuição, a variável do lado direito da igualdade perde o valor.	Em $m = n$, o aluno acha que m fica com o valor de n e n fica sem valor.
O aluno acha que o nome da variável é usado pelo programa para "gerar" um valor coerente para ela.	Se o programa tem uma variável com o nome *media*, então o programa conseguirá automaticamente atribuir a média de uma lista de inteiros a essa variável.
O estudante acredita que uma variável pode ter mais de um valor simultaneamente.	O aluno define as instruções: $a = 3$; e $a = 4$, e depois executando $b = a + a$, ele imagina que o valor de b será 7.
O aluno acredita que algumas linhas do programa são executadas em paralelo e ficam válidas durante toda a execução do programa.	O aluno define as seguintes instruções: $x = y + z$, $y = 3$ e $z = 2$ e imagina que o valor de x será 5 em qualquer momento da execução do programa
O aluno acredita que as variáveis são inicializadas automaticamente.	Na sentenças: `int x`, $x = x + 1$, o aluno imagina que x terá um valor inicial padrão.
Caso uma sentença do tipo *if* não tenha *else*, o aluno pensa que o programa para, caso a condição desse *if* seja falsa.	
O aluno pensa que tanto o ramo *then* quanto o ramo *else* de uma seleção são sempre executados.	

O aluno pensa que as sentenças do ramo *then* de uma seleção são sempre executadas	
O aluno pensa que as instruções que estão após o corpo de um condicional sem *else* serão executadas somente se a condição da seleção seja falsa.	
O aluno pensa que o laço será finalizado na linha que altera o valor de uma variável que faz com que a condição da expressão seja falsa. Nesse caso, as sentenças seguintes dentro do laço não seriam executadas.	
O aluno pensa que as variáveis na condição de um laço *for* não possuem valor no corpo do laço.	
O aluno pensa que o intervalo de valores da variável de controle do laço restringe os valores que as variáveis no corpo do laço podem assumir.	
Falha em que o laço é executado uma vez a mais ou uma vez a menos. Esse problema normalmente ocorre quando o aluno inicia a variável de controle errado ou usa o operador do teste de condição errado.	
O aluno pensa que uma sentença após o laço faz parte do corpo do laço. O aluno não sabe dizer onde o laço inicia e onde ele termina.	
O aluno pensa que somente a última linha do corpo do laço será executada várias vezes, e as primeiras somente uma vez.	
O aluno pensa que definindo um bloco ele funcionará como laço.	
O aluno acredita que o computador tem inteligência para controlar quando o laço termina.	
O aluno pensa que uma variável pode ter mais que um valor e assim tratar uma seleção como se fosse um laço.	
O aluno acredita que para processar n números lidos é preciso usar dois laços em seqüência. O primeiro laço é responsável por fazer a leitura dos valores e o outro laço por fazer o processamento.	

Tabela 2.1: Erros conceituais lógicos, tipicamente cometidos por aprendizes de programação.

Exemplos de falhas que serão tratadas pelo sistema de depuração automática proposto (HPD). Na Tabela 2.2, mostramos exemplos de falhas que o nosso sistema de depuração automática será capaz de detectar. A primeira coluna dessa tabela é o identificador da falha, a segunda coluna mostra uma descrição da falha e na terceira coluna são apresentados os exemplos de como essas falhas ocorrem. As falhas apresentadas foram divididas de acordo com a forma com que elas se manifestam, ou seja, em:

- *inicializações*: falhas em inicializações que utilizam uma constante ou uma única variável do lado direito do sinal de atribuição;
- *expressões e operadores*: geradas por inversão de operandos, uso do operador errado, uso de constante ou variável errada na expressão;
- *construções em laços ou seleções (if-then-else)*: falhas relacionadas ao uso do laços ou seleções;
- *relacionadas a métodos*: falhas que aparecem em métodos ou na chamada de métodos.

Falhas em inicializações		
Identificador	**Descrição**	**Exemplo**
F1	inicialização de variável com valor constante incorreto	• `x = 3` ao invés de `x = 1`;
F2	inicialização de variável com o valor de outra variável incorreta	• `x = k` ao invés de `x = y`
Falhas em expressões e operadores		
Identificador	**Descrição**	**Exemplo**
F3	operandos em posições trocadas	• `a / b` ao invés de `b / a`
F4	uso de variável errada em expressão	• `a + b` ao invés de `a + c`
F5	uso de constantes erradas em expressão	• `x + 3` ao invés de `x + 2`
F6	expressão lógica com variáveis incorretas	• `(a && (b || !c))` ao invés de `(d && (b || !c))`
F7	expressão lógica com constantes incorretas	• `(a && (false || !c))` ao invés de `(a && (false || !c))`

F8	uso de operações envolvendo números inteiros que resultariam em números de ponto flutuante	int a = 5;
		int b = 2;
		int c = a / b;

F9	uso incorreto dos operadores de incremento/decremento	• ++i ao invés de i++ ou vice-versa; • --i ao invés de i-- ou vice-versa.
F10	uso incorreto do operador unário ! (NOT) e em expressão lógica	• !<expr. lógica> ao invés de <expr. lógica>.
F11	uso incorreto dos seguintes operadores lógicos: $\{<, >, ==, <=, >=, \|\|, \&\&\}$	• a < b ao invés de a <= b; • a > 3 ao invés de a >= 3; • a != b ao invés de a == b;
Laços ou seleções (*if-then-else*)		
Identificador	**Descrição**	**Exemplo**
F12	uso incorreto de operador na condição de seleções e laços	• if (x < y) ao invés de if (x <= y)
F13	inversões nos conjuntos de sentenças de componentes de seleções	if (a < b) <ações1> else <ações2> ao invés de: if (a < b) <ações2> else <ações1>

F14	atribuições incorreta no corpo de laços e seleções	`a = x + 1;`
		`b = 3;`
		`...`
		`while(k < 10) {`
		`    ...`
		`    c = a + b;`
		`    \\ o correto é c = b + d;`
		`    ...`
		`}`

F15	atualização incorreta da variável de controle do laço	`int v;`
		`  ...`
		`while (p < max) {`
		`  ...`
		`  p = p + 1;`
		`  \\ o correto é p = p + 2; }`

Falhas relacionadas a métodos		
Identificador	**Descrição**	**Exemplo**
F16	passagem de parâmetro incorreto na chamada de métodos	`int x = soma(b, c);` `\\ o correto é int x = soma(a, b);`
F17	passagem de parâmetros invertidos na chamada de métodos	`int x = soma(b, a);` `\\ o correto é int x = soma(a, b);`

Tabela 2.2: Tipos de falhas que podem ser detectadas pelo sistema de depuração automática proposto nessa dissertação.

2.2 Ensino/Aprendizagem de programação auxiliada por padrões

Os padrões elementares são estratégias para resolver problemas de programação, que podem ser utilizadas para auxiliar alunos novatos no aprendizado de programação, sendo seu uso recomendado por educadores de programação [35].

Segundo pesquisas referentes a teorias cognitivas sobre o aprendizado de programação [25], os programadores experientes buscam soluções de novos problemas recorrendo a soluções anteriores. Como aprendizes de programação ainda não possuem experiências anteriores na resolução de problemas, quando eles são expostos a um problema que deve ser solucionado, a única alternativa é tentar construir a solução completa desde o início.

Esse problema na aprendizagem de programação já tinha sido analisado no trabalho de So-

loway [38], que utilizava um modelo de planos e metas para tentar explicar o processo de raciocínio de programadores experientes e como isso poderia ser aplicado na aprendizagem de programação. Porém, planos e metas são internos ao sistema, não permitindo assim que os alunos tenham acesso a eles. Com o trabalho de East et. al. [17], foi proposto o uso de padrões para auxiliar alunos novatos a aprenderem como programar.

Em trabalhos posteriores de pesquisadores da área de educação [43, 1, 6, 44], foi proposto um conjunto de padrões que podem ser usados dentro de uma metodologia de ensino, para fornecer ao aluno uma variedade de estratégias gerais de resolução de problemas de introdução à programação. A partir desse conjunto de padrões é possível organizar uma disciplina que permita ao aprendiz, além de aprender uma linguagem de programação, aprender também várias estratégias gerais de resolução de problemas, apresentadas em sala de aula.

Um padrão elementar é descrito por um conjunto de atributos que juntos formam a *documentação do padrão*. Essa documentação pode ser usada pelos alunos para auxiliá-los a compreender em quais situações um determinado padrão deve ou não ser empregado, que efeitos decorrem do uso de um padrão, entre várias outras informações. Não existe uma única forma de se documentar os padrões, e na literatura é possível encontrar vários exemplos diferentes [20, 7, 43, 1, 6]. Nesse trabalho, utilizaremos um formato para documentar os padrões derivado daquele utilizado no trabalho de Bergin [6]. A Tabela 2.3 mostra alguns dos atributos que podem ser usados para documentar um padrão elementar e suas respectivas descrições.

No Apêndice A, são apresentados os padrões elementares utilizados nesse trabalho. Uma descrição detalhada sobre padrões de projeto em geral pode ser encontrada em [20,7], e uma referência detalhada sobre padrões elementares pode ser encontrada em [1, 6, 15].

2.3 Depuração de programas

Nessa seção, apresentamos brevemente os principais conceitos referentes à depuração de programas.

Segundo a definição do IEEE [24], depuração de software é o processo de detectar, localizar e corrigir falhas em um programa de computador. Apesar dessa tarefa ser considerada muito comum na construção de programas, em muitos casos ela pode ser complexa e requerer um longo tempo para que a falha seja encontrada. Por esse motivo, construir um depurador automático de programas não é uma tarefa trivial e não oferece garantias de completude, isto é, não garantem encontrar uma falha existente.

Atualmente, existem muitas ferramentas de depuração que disponibilizam funcionalidades como: acompanhar o fluxo de execução do programa, inserir pontos de interrupção durante a execução de programas, inspecionar o valor de variáveis nos pontos de interrupção, execução passo-a-passo do programa, entre outras.

Atributo	**Descrição**
Intenção do padrão	Define a finalidade do padrão e qual tipo de problema ele resolve.
Intenção pedagógica	Informação definida pelo professor que expressa qual estratégia é esperada que seja adquirida pelo aluno com a compreensão desse padrão.
Motivação	Define um cenário de uso do padrão.
Estrutura da solução 1: sintaxe	Descrição em pseudocódigo da sintaxe do padrão e os *metadados* que ele manipula.
Estrutura da solução 2: semântica	Descreve o comportamento do padrão, incluindo o fluxo de execução.
Aplicabilidade / Objetivo	Descreve as situações em que o padrão pode ser aplicado.
Pré-requisitos	Define os conceitos que o aluno deve saber para que ele possa compreender e usar esse padrão.
Implementação	Apresenta sugestões, restrições ou mesmo cuidados que o aluno deve tomar durante a implementação do padrão.
Código de exemplo	Apresenta um exemplo de como o padrão é implementado em uma ou mais linguagens de programação.
Usos conhecidos	Cita algumas situações ou domínios nos quais o padrão pode ser utilizado (podendo, inclusive, mostrar exercícios nos quais o padrão pode ser aplicado).
Padrões relacionados	Cita outros padrões elementares diretamente relacionados com esse padrão.

Tabela 2.3: Atributos usados para documentar um padrão elementar [15].

Uma outra forma de encontrar falhas em programas, que tem sido muito explorada atualmente, é através de *casos de teste*. Um caso de teste define um conjunto de entradas e saídas esperadas para um determinado programa e ao ser executado é verificado se o resultado obtido no programa é o mesmo especificado nas saídas dos casos de teste.

Em se tratando de aprendizes de programação, o processo de depuração normalmente recomendado por educadores é chamado de *teste de mesa*, na qual os aprendizes simulam a execução de cada uma das instruções do programa utilizando lápis e papel. A opção por utilizar um processo desse tipo é que os alunos tenham a oportunidade de reconhecer suas próprias falhas e também entender melhor os detalhes da execução do programa. A técnica para encontrar falhas que apresentamos nesse trabalho é uma forma de simulação feita pelo computador, capaz de detectar as falhas durante esse processo. A diferença entre o teste de mesa e o processo de depuração baseada em modelos é que, além de simular a execução do programa da entrada para a saída, esse processo

também é capaz de propagar os valores das variáveis da saída para a entrada.

Além das abordagens citadas, existem também os depuradores automáticos de programas, que são ferramentas capazes de analisar o programa e encontrar as possíveis falhas. Podemos considerar dois grupos distintos de depuradores automáticos de programas [16, 15]:

- *Tutores Inteligentes de Programação*, que são sistemas utilizados na aprendizagem de programação e geralmente voltados para uma tarefa muito específica, ou para uma linguagem muito reduzida. Nesse tipo de sistema, supõe-se que o aprendizado ocorra através da comunicação do aluno com o tutor.
- *Sistemas de Depuração de Programas*, que são geralmente utilizados para encontrar falhas em programas desenvolvidos por programadores experientes. Nesse tipo de sistema, não é necessário que exista uma especificação completa do programa.

Para fazer a depuração dos programas, esses dois grupos de depuradores automáticos utilizam certas técnicas de correção automática, a saber:

- *verificação com respeito à especificação*, no qual existe uma especificação formal para o programa, que é confrontada com o programa implementado, apontando as diferenças (falhas);
- *verificação com respeito ao conhecimento da linguagem*, que tenta encontrar pontos suspeitos no programa que não estão de acordo com o conhecimento da linguagem (como por exemplo, tentar usar variáveis sem inicializá-las, não incrementar a variável de controle de um laço, entre outras);
- *filtragem com respeito aos sintomas*, que, baseado nos resultados obtidos na saída do programa, é capaz de presumir que certas partes do programa não podem conter falhas.

A técnica de Diagnóstico baseado em Modelos [14, 36] é um exemplo da técnica de filtragem com respeito aos sintomas, que também é usada para fazer depuração de programas [26, 27, 30, 29]. Essa técnica corresponde ao estado da arte das técnicas de IA para fazer a depuração automática e, atualmente, são utilizadas para encontrar falhas em programas escritos em Java ou C++. Nessa abordagem, um modelo lógico é derivado do programa e, com base em entradas e saídas definidas por um caso de teste, são apontadas as linhas do programa que contêm falhas. Um grande diferencial dessa abordagem é que não é necessário existir uma especificação formal do programa para fazer a depuração.

2.4 Trabalhos correlatos

A seguir, são apresentados os dois trabalhos da área de depuração automática de programas para o ensino de programação que mais se relacionam a proposta dessa dissertação. Note que são

mencionadas somente duas ferramentas, pois existem poucos trabalhos que utilizam a depuração de programas para promover o aprendizado.

2.4.1 PROUST

PROUST [25] é uma ferramenta voltada para o aprendizado de programação, capaz de reconhecer *metas* e *planos* no programa do aluno e confrontá-los com metas e planos previamente definidos na especificação do problema. A meta é uma tarefa relacionada ao problema que o aluno tenta resolver, enquanto que o plano é um procedimento ou uma estratégia utilizada para realizar uma tarefa que o aluno necessita para alcançar uma meta.

Para que o PROUST possa reconhecer as falhas no programa do aluno, é necessário que ele receba uma descrição do problema especificando as metas e como elas estão relacionadas. O relacionamento das metas é feito de forma hierárquica, definindo metas e submetas. Para cada uma dessas metas e submetas são associados os planos que podem ser utilizados para alancá-las.

A partir do programa do aluno, o PROUST utiliza uma biblioteca de planos previamente definidos para tentar identificar quais os planos que o aluno utilizou em seu programa. Se for encontrada alguma divergência ao confrontar os planos reconhecidos no programa do aluno com os planos definidos na especificação do problema, então uma falha no programa é reconhecida e reportada ao aluno.

Note que ao reconhecer um plano no programa do aluno, indiretamente também é identificada qual a sua intenção durante a construção do programa. Assim, ao reportar a falha ao aluno, o que se espera é que ele consiga entender a diferença entre a sua intenção e o que foi construído no programa. O Exemplo 2.1, mostra um problema que utilizaremos para mostrar alguns planos detectadas pelo PROUST.

Exemplo 2.1 - Problema da média aritmética. Construa um programa que resolva o problema da *Média Aritmética*, descrito da seguinte forma: *Calcular a média aritmética de uma sequência de números inteiros lidos da entrada padrão. Interprete o número* 99999 *como sendo o final da sequência (esse valor não deve ser considerado para o cálculo da média). Somente variáveis inteiras devem ser utilizadas.* □

O Programa 2 [25] foi construído por um aluno para resolver o problema da *Média Aritmética*. Os números apresentados em parênteses ao lado esquerdo de cada linha indicam os planos nos quais essas linhas pertencem, conforme reconhecidos pelo PROUST. Os seguintes planos foram reconhecidos no Programa 2:

Programa 2 Programa que resolve o problema de *Média Aritmética*. Ao lado de cada linha estão relacionados os planos nos quais ela faz parte.

```
                    1   public static void main(String[] args) {
                    2     Scanner s = new Scanner(System.in);
                    3     float soma;
                    4     float valor;
                    5     float media;
                    6     float cont;
                    7     padrão.
(1)                 8     cont = 0;
    (2)             9     soma = 0;
                    10    /* o método s.nextFloat() é utilizado para obter
                    11      um valor inteiro da entrada padrão */
        (3)         12    valor = s.nextFloat();
        (3)         13    while (valor != 99999) {
    (2) (3)         14      soma = soma + valor;
(1)                 15      cont = cont + 1;
        (3)         16      valor = s.nextFloat();
        (3)         17    }
                    18
            (4)     19    if (cont > 0) {
            (4)     20      media = soma / cont;
            (4)     21      System.out.println(media);
            (4)     22    } else
            (4)     23      System.out.println("Entrada inválida!");
                    24  }
```

(1) *Counter variable plan*, responsável por atribuir valor a uma variável responsável por fazer uma contagem;

(2) *Running total variable*, responsável por manter valores relacionados ao domínio do problema (e.g., peso, idade, somatória, entre outras);

(3) *Running total loop plan*, que define um método para computar o valor que é armazenado em uma variável definida pelo Running total variable plan;

(4) *Valid result skip guard plan*, que é responsável por controlar quando um trecho do código deve ou não ser executado de acordo com uma condição.

Apesar de muito citado na literatura, o PROUST tem as seguintes limitações:

- é necessário que exista um especialista para especificar os planos que serão armazenados na biblioteca;
- devido ao fato dos planos serem armazenados numa biblioteca, estes podem não ser suficientes

para cobrir todas as soluções possíveis para um determinado problema. Caso isso aconteça, o PROUST não será capaz de reconhecer as falhas;

- o sistema foi desenvolvido para encontrar falhas em programas escritos na linguagem Pascal.
- não existe a ideia de estratégias gerais que possam ser apresentadas ao aluno em sala de aula, o que diminui a chance de identificá-las no programa do aluno.

2.4.2 ProPAT

O ProPAT [3, 15] é um ambiente voltado para a aprendizagem de programação com padrões elementares. Nesse ambiente, o aluno tem a possibilidade de escolher o padrão elementar que deseja utilizar e, com o auxílio da ferramenta, inserir um esqueleto do padrão no editor. A Figura 2.1 mostra uma visão do ambiente ProPAT, que permite inserir padrões elementares no programa do aluno.

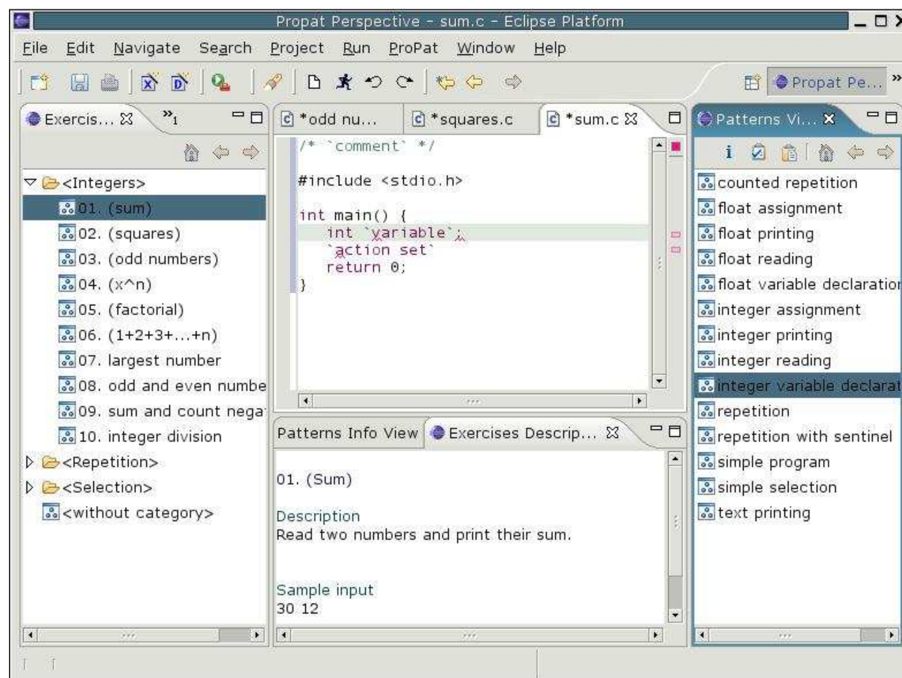


Figura 2.1: Visão de padrões do ProPAT.

O ProPAT também disponibiliza um depurador automático de programas, que utilizada a técnica de depuração de programas baseada em modelos (MBSD) para encontrar as falhas no programa do aluno. Apesar de não construir o modelo do programa utilizando padrões elementares, é possível fazer a comunicação das falhas com o aluno em termos de padrões.

Outra característica importante no ProPAT é a ordenação feita para apresentar as hipóteses de falha. Com base no fluxo de execução de diversos casos teste, são atribuídas probabilidades de falha a cada hipótese, que são ordenadas e apresentadas as hipóteses mais prováveis primeiro.

Por utilizar a técnica de MBSD na implementação do depurador automático, o ProPAT tem as mesmas limitações que foram citadas na Seção 1.1. Outra razão que consideramos limitantes no uso desse ambiente, é o fato dele ser capaz de reconhecer somente os programas escritos na linguagem C.

Essa dissertação tem o objetivo de estender o sistema ProPAT para: tratar programas na linguagem JAVA, estender a técnica de depuração para tratar componentes abstratos e usar a plataforma Dr. Java com o depurador automático.

2.5 Considerações finais

Nesse capítulo, apresentamos os principais conceitos relacionados ao processo de aprendizagem de programação. Foram introduzidos os conceitos de falhas típicas de programação e uma taxionomia de falhas; os padrões elementares e como esses padrões podem ser utilizados no processo de ensino/aprendizagem de programação; as principais técnicas de depuração de programas, incluindo algumas abordagens para fazer a depuração automática de programas e, por fim, os principais trabalhos correlatos.

No próximo capítulo, será apresentado o conceito de Diagnóstico baseado em Modelos, que constitui a base para todas as técnicas de Depuração de Programas baseada em Modelos e também da abordagem que propomos nesse trabalho.

Capítulo 3

Diagnóstico Baseado em Modelos

Este capítulo apresenta um método de diagnóstico de sistemas chamado de *Diagnóstico Baseado em Modelos* (*Model Based Diagnosis* - MBD), uma importante técnica desenvolvida pela comunidade de Inteligência Artificial para detectar falhas em sistemas. Serão apresentados: o modelo conceitual que divide a tarefa de diagnóstico em sub-tarefas e o Algoritmo de Reiter (*Minimal Hitting Set*) [36], utilizado em uma das sub-tarefas de diagnóstico. Ao final, apresentamos uma discussão sobre a complexidade do problema de diagnóstico.

3.1 Conceitos básicos de Diagnóstico Baseado em Modelos

Diagnosticar é a tarefa de inferir hipóteses sobre as partes falhas de um sistema que apresenta comportamento discrepante do esperado. Para realizar essa tarefa, é possível utilizar duas abordagens fundamentais [5]:

- *Diagnóstico baseado em heurísticas*, que usa a experiência acumulada de um especialista em diagnósticos anteriores de um determinado sistema, para construir uma base de regras (*heurísticas*), normalmente na forma de regras do tipo *causa-efeito*. Apesar de ser aplicável em diversas situações, o diagnóstico baseado em heurísticas apresenta as seguintes desvantagens: (1) dependência do domínio do problema, (2) limitação do conhecimento do especialista e (3) dificuldade de se construir explicações a partir do processo de inferência [5].

- *Diagnóstico baseado em modelos (MBD)* [36, 14], que raciocina sobre modelos que descrevem o comportamento correto do sistema. De um lado, está o sistema físico, possivelmente falho, cujo comportamento pode ser observado; do outro lado, está o modelo que é usado para fazer predições a respeito do comportamento correto do sistema. A diferença entre uma observação e uma predição é chamada de *discrepância*, enquanto uma equivalência é chamada de *concordância*. Discrepâncias e concordâncias são utilizadas para encontrar o diagnóstico para o sistema [5] (Figura 3.1). As técnicas de MBD são independentes do domínio do problema e são capazes de encontrar falhas múltiplas. Como é necessário que exista um modelo descrevendo o comportamento correto do sistema, essa abordagem somente é aplicável

a sistemas construídos pelo homem.

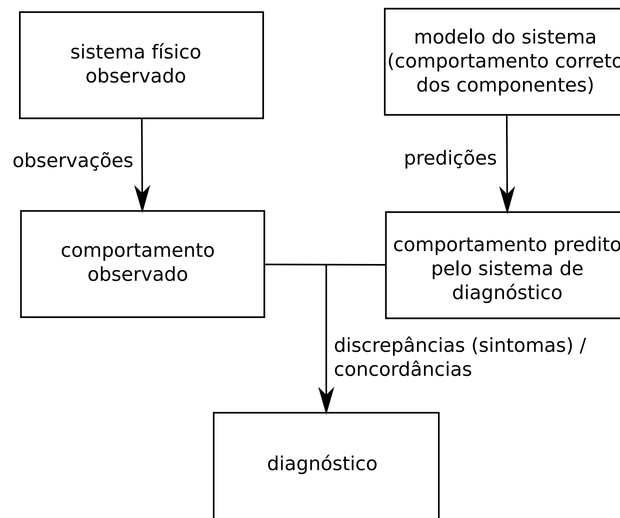


Figura 3.1: O diagnóstico é construído com base nas discrepâncias e concordâncias encontradas ao se comparar as observações e as predições do comportamento do sistema [5].

Os sistemas podem ser descritos por diversos tipos de modelos que, em geral, são classificados como:

- *modelos centrados em componentes*, que descrevem o sistema em função de seus componentes, suas conexões e o comportamento desses componentes;
- *modelos causais*, que descrevem o sistema em termos de relações entre estados do sistema, na forma de *causa-efeito*, isto é, estados causam outros estados como efeito;
- *modelos funcionais*, que descrevem o sistema em termos de funções e subfunções.

De acordo com o tipo de inferência utilizada, a tarefa de diagnóstico pode ser classificada como: *abdutiva*, na qual a solução do diagnóstico tem como consequência lógica o conjunto de todas as observações ou; *baseada em consistência* [14, 36], na qual a solução do diagnóstico é consistente com as observações [5]. No diagnóstico abdutivo, um diagnóstico é um conjunto minimal de suposições de falhas que explica as observações [9]. No diagnóstico baseado em consistência, dado um modelo do sistema a ser diagnosticado, um diagnóstico é um conjunto minimal de suposições de falhas (componentes anormais) tal que as observações são consistentes com todos os demais componentes normais. Nesse trabalho, abordaremos somente MBD baseado em consistência, com modelos centrados em componentes, pois esse modelo é o que melhor se adapta à representação que necessitamos.

3.2 Modelos centrados em componentes

Os dois tipos de modelos centrados em componentes mais usados para diagnóstico são: o *modelo comportamental*, que descreve o comportamento correto de cada componente em função de suas entradas e saídas e; o *modelo estrutural*, que descreve as conexões entre os componentes. Existem outros modelos centrados em componentes que também podem ser usados num processo de diagnóstico, entre eles: o *modelo topológico*, que descreve a vizinhança (proximidade) dos componentes e o *modelo de falhas*, que descreve o comportamento falho dos componentes.

Um sistema automático de diagnóstico, ou simplesmente sistema de diagnóstico, é capaz de raciocinar sobre o modelo comportamental utilizando dois tipos de axiomas [5]:

- *axiomas de simulação*, que permitem predizer as saídas de um componente, dadas as suas entradas. Esses axiomas simulam o comportamento correto do componente;
- *axiomas de satisfação de restrições*, que descrevem um comportamento que não acontece no componente real, mas permite fazer inferências válidas. Por exemplo, em um componente com duas entradas e uma saída, é possível inferir o valor de uma das entradas, a partir da outra entrada e da saída (simulação reversa).

Exemplo 3.1 - Circuito eletrônico. A Figura 3.2 mostra um sistema físico, que corresponde a um circuito eletrônico contendo uma ou mais falhas. Este circuito é composto de três componentes multiplicadores, $M1$, $M2$ e $M3$, dois adicionadores, $A1$ e $A2$, cinco entradas: A, B, C, D e E e duas saídas: F e G. Nas saídas dos multiplicadores são definidos os pontos: X, Y e Z. Além disso, as entradas são: $A = 3$, $B = 2$, $C = 2$, $D = 3$ e $E = 3$, e as saídas: $F = 10$ e $G = 12$. □

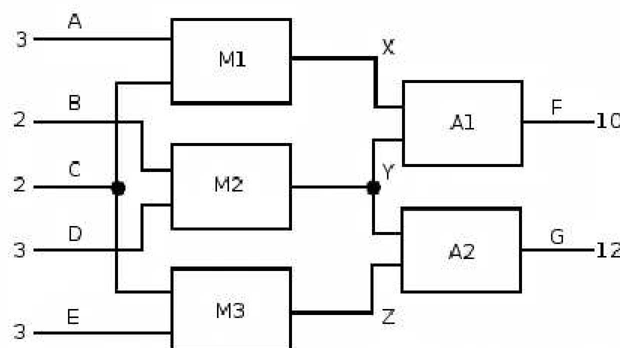


Figura 3.2: Circuito de exemplo [21].

Pelos axiomas de simulação é possível inferir os valores $X = 6$, $Y = 6$, $Z = 6$, $F = 12$ e $G = 12$, mostrando que há uma discrepância entre o valor medido em F e o valor gerado pela simulação. Essa discrepância é uma observação de mal-funcionamento do sistema, que chamaremos de *sintoma de mal-funcionamento*, ou simplesmente de *sintoma*.

3.3 Tarefas do processo de diagnóstico

Um método básico de diagnósticos decompõe a tarefa de diagnóstico nas seguintes sub-tarefas [5]:

1. *Detecção de sintomas*, responsável por verificar quais observações diferem do esperado (discrepantes), caracterizando os *sintomas* de mal-funcionamento do sistema.

2. *Geração de hipóteses*, responsável por gerar as hipóteses que explicam as manifestações dos sintomas. Essa tarefa é composta das seguintes sub-tarefas:

   (a) *Busca por contribuintes*, que encontra os *conjuntos de contribuintes*. Contribuintes são componentes envolvidos em uma predição discrepante de uma observação. Para cada predição discrepante, encontra-se um conjunto de contribuintes.

   (b) *Transformação de conjunto de contribuintes em hipóteses*, que gera os *conjuntos de hipóteses* a partir dos conjuntos de contribuintes. Uma hipótese é composta de um ou mais componentes que quando simultaneamente falhos explicam todos os sintomas detectados.

   (c) *Predição baseada em filtragem*, responsável por reduzir o conjunto de hipóteses usando somente o conhecimento já existente sobre o sistema (não requer observações adicionais), como, por exemplo, o conhecimento do comportamento de falha de um determinado componente.

3. *Discriminação de hipóteses*, que utiliza observações adicionais para reduzir ainda mais o conjunto de hipóteses e, eventualmente, chegar à verdadeira causa de falha. Por exemplo, escolhendo-se pontos estratégicos do sistema para fazer novas observações (pontos X, Y e Z da Figura 3.2).

A Figura 3.3 apresenta o relacionamento entre tarefas e sub-tarefas do diagnóstico, segundo um método básico de decomposição dessa tarefa. É interessante notar que, segundo Benjamins [5], esse método foi usado pela maioria do sistemas de diagnóstico descritos na literatura.

3.4 Formalização do problema de diagnóstico

Nesta seção, mostramos a formalização para o problema de diagnóstico conforme apresentado em [36]. Essa formalização trata especificamente da sub-tarefa de geração de hipóteses (retângulo destacado na Figura 3.3).

Definição 3.1. *Um sistema é dado por um par* $\langle SD, COMP\rangle$*, sendo* SD *a descrição do sistema e* $COMP$ *um conjunto finito de constantes representando os componentes do sistema. A descrição do sistema é feita através de um conjunto de sentenças em lógica de primeira ordem, com igualdade*

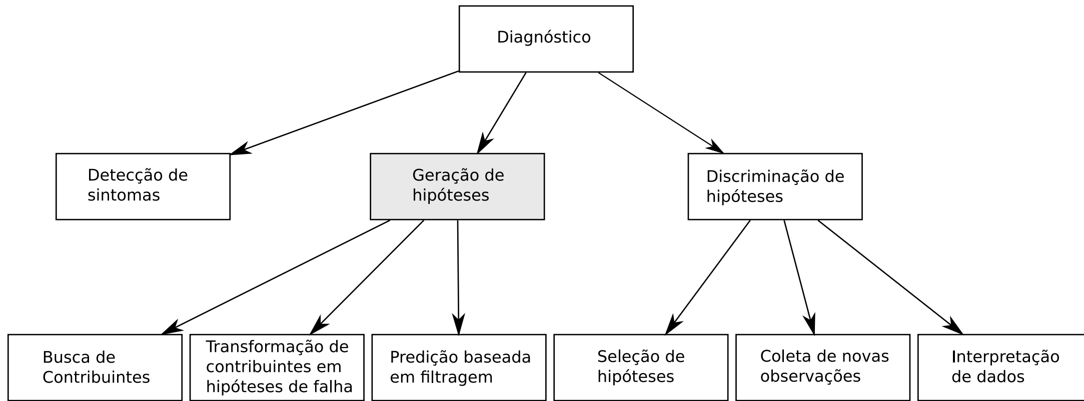


Figura 3.3: Decomposição do processo de diagnóstico em sub-tarefas [5].

e funções, descrevendo o comportamento dos componentes do sistema (modelo comportamental) e as conexões entre os componentes do sistema (modelo estrutural), com suas entradas e saídas.

Um sistema real pode ter componentes nos quais os valores associados às suas entradas e/ou saídas podem ser medidos. Essas medições constituem as observações no sistema que são utilizadas em praticamente todas as sub-tarefas do diagnóstico.

Definição 3.2. *Seja OBS um conjunto de observações feitas sobre os valores das conexões do sistema físico, representadas por sentenças em lógica de primeira ordem (fatos). A tripla $\langle SD, COMP, OBS\rangle$ denota um problema de diagnóstico com o sistema $\langle SD, COMP\rangle$ e observações OBS.*

Exemplo 3.2 - Modelo de sistema. O sistema representando o circuito do Exemplo 3.1 pode ser descrito por $\langle SD, COMP\rangle$, sendo $COMP = \{M_1, M_2, M_3, A_1, A_2\}$, SD (modelo comportamental e estrutural) e OBS (observações) dados pelas seguintes sentenças[1]:

Modelo comportamental

$$adder(x) \wedge ok(x) \rightarrow add(in_1(x), in_2(x), out_1(x))$$
$$multiplier(x) \wedge ok(x) \rightarrow mult(in_1(x), in_2(x), out_1(x))$$

$$multiplier(M_1)$$
$$multiplier(M_2)$$
$$multiplier(M_3)$$
$$adder(A_1)$$
$$adder(A_2)$$

Modelo estrutural

[1]Considere que os axiomas de operações sobre números inteiros estejam corretamente definidos.

$out_1(M_1) = in_1(A_1)$
$out_1(M_2) = in_2(A_1)$
$out_1(M_2) = in_1(A_2)$
$out_1(M_3) = in_2(A_2)$

Observações

$in_1(M_1) = 3$
$in_2(M_1) = 2$
$in_1(M_2) = 2$
$in_2(M_2) = 3$
$in_1(M_3) = 2$
$in_2(M_3) = 3$
$out_1(A_1) = 10$
$out_1(A_2) = 12$

□

No modelo comportamental desse circuito, o predicado unário ok é usado para representar o comportamento de um componente: $ok(C)$ indica que o componente $C \in COMP$, funciona corretamente, enquanto $\neg ok(C)$ indica que o componente C está falho. Os predicados *adder* e *multiplier* indicam os tipos dos componentes de adição e multiplicação, respectivamente. Nesse caso, M_1, M_2 e M_3 são multiplicadores (representados pelos fatos: $multiplier(M_1)$, $multiplier(M_2)$ e $multiplier(M_3)$, respectivamente), enquanto A_1 e A_2 são adicionadores (representados pelos fatos $adder(A_1)$ e $adder(A_2)$, respectivamente). O predicado $add(i1, i2, out)$ representa uma operação de adição ($i1+i2 = out$) e $mult(i1, i2, out)$ a operação de multiplicação ($i1*i2 = out$). $in_i(x)$ representa uma função que devolve o valor da $i-ésima$ entrada do componente x, e $out_i(x)$ representa uma função que calcula o valor da $i-ésima$ saída do componente x. O modelo estrutural descreve como os componentes estão conectados em função das suas entradas e saídas. Por exemplo, $out_1(M_1) = in_1(A_1)$ indica que a saída 1 do componente M_1 está conectada à entrada 1 do componente A_1. Além disso, são declaradas as observações das entradas e saídas com fatos representando igualdade entre as funções in/out e valores constantes.

Definição 3.3. *Um problema de diagnóstico $P_{MBD} = \langle SD, COMP, OBS\rangle$ consiste em encontrar um conjunto de componentes $\Delta \subseteq COMP$ que, quando considerados falhos, expliquem as observações (OBS).*

Seja $P_{MBD} = \langle SD, COMP, OBS\rangle$ um problema de diagnóstico. Se o sistema real, representado por $\langle SD, COMP\rangle$, possui um ou mais componentes falhos, então

$$SD \cup OBS \cup \{ok(C)|C \in COMP\} \tag{3.1}$$

é inconsistente[2], ou seja, a suposição de que todos os componentes do sistema possuem um comportamento correto permite inferir valores discrepantes dos observados.

Dado um conjunto maximal de componentes falhos $\Delta \subseteq COMP$, a consistência da Fórmula 3.1 pode ser restabelecida removendo as cláusulas (fatos): $\{ok(C)|C \in \Delta \subseteq COMP\}$. Uma forma simples de restabelecer a consistência dessa fórmula é fazendo $\Delta = COMP$, indicando que todos os componentes do sistema apresentam falha. Essa abordagem, porém, é pouco informativa e requer, no pior caso, que todos os componentes do sistema sejam medidos ou substituídos, tornando esse resultado praticamente desprezível. Para evitar esse tipo de situação, é adotado o *Princípio de Parcimônia*[3], e somente as hipóteses de falha minimais devem ser consideradas.

Definição 3.4. *Um diagnóstico para $\langle SD, COMP, OBS\rangle$, que obedece o princípio de parcimônia, é um conjunto minimal $\Delta \subseteq COMP$, tal que:*

$$SD \cup OBS \cup \{ok(C)|C \in COMP - \Delta\} \cup \{\neg ok(C)|C \in \Delta\} \tag{3.2}$$

consistente.

Segundo a Definição 3.4, é possível encontrar Δ fazendo uma enumeração de todos os subconjuntos $\Delta' \subseteq COMP$, e verificando a consistência de

$$SD \cup OBS \cup \{ok(C)|C \in COMP - \Delta'\} \tag{3.3}$$

Note que podem ser encontrados diversos subconjuntos $\Delta' \subseteq COMP$ que tornam a Fórmula 3.3 consistente, ou seja, podem existir diversos diagnósticos distintos (hipóteses da falha). Apesar dessa abordagem de enumerar e verificar permitir que todos os diagnósticos para um sistema sejam encontrados, ela é muito ineficiente. De Kleer [14] apresentou uma forma de chegar ao mesmo resultado utilizando conjuntos de contribuintes[4].

Definição 3.5. *Um conjunto de contribuintes para $\langle SD, COMP, OBS\rangle$ é um conjunto $CO \subseteq COMP$ tal que:*

$$SD \cup OBS \cup \{ok(C)|C \in CO\} \tag{3.4}$$

é inconsistente.

Definição 3.6. *Seja F uma família de conjuntos. Um* conjunto de corte *(hitting set) é um conjunto $H \subseteq \bigcup_{S \in F} S$ tal que $H \cap S \neq \emptyset$ para todo $S \in F$. Um conjunto de corte é minimal se e somente se não existir subconjunto próprio dele que também seja um conjunto de corte.*

[2] Supõe-se que SD é consistente. Caso contrário, o problema de diagnóstico não tem solução.

[3] O princípio de parcimônia diz que perante diversas justificativas, devemos escolher aquela que for a mais simples (a justificativa que requer o menor número de deduções). Esse princípio também é conhecido como *Occam's Razor*.

[4] Em [14], o conjunto de contribuintes é chamado de *conjunto conflito*.

O seguinte teorema forma a base para encontrar o diagnóstico:

Teorema 3.1. $\Delta \subseteq COMP$ *é um diagnóstico para* $\langle SD, COMP, OBS\rangle$ *se e somente se* Δ *é um conjunto de corte minimal para a coleção de conjuntos de contribuintes de* $\langle SD, COMP, OBS\rangle$ [36].

Informalmente, um conjunto de contribuintes é um conjunto de componentes que tendo seus comportamentos definidos como corretos permitem predizer valores discrepantes dos valores observados, caracterizando um ou mais sintomas. Durante a busca de contribuintes (sub-tarefa da geração de hipóteses), ao assumir que um componente está falho, este componente não poderá ser usado para predizer valores. Suponha que um componente C pertença a um conjunto de contribuintes CO_0 e C esteja falho. Se nessa condição ainda forem verificadas outras discrepâncias, então, deve existir ao menos um conjunto de contribuintes $CO_1 \neq CO_0$, tal que $C \notin CO_1$, responsável pelas discrepâncias. Repetindo o processo de escolher um componente C_1 desse conjunto de contribuintes encontrado, supor que C_1 está falho e verificar se ainda existem discrepâncias no sistema, permite que sejam obtidas todas as hipóteses de falha, sendo cada uma composta por componentes escolhidos de cada um dos conjuntos de contribuintes. Pode-se dizer que cada hipótese de falha é responsável por justificar todos os sintomas manifestados no sistema e justificados pelos conjuntos de contribuintes.

Note que, caso sejam encontrados dois conjuntos de contribuintes, CO_i e CO_j, tal que $CO_j \subset CO_i$, então, nenhum componente $C \in (CO_i - CO_j)$ pode aparecer nas hipóteses de falha (exceto os casos onde exista um terceiro conjunto de contribuintes, CO_k, no qual $C \in CO_k$). Isso ocorre devido ao fato de que, mesmo supondo que o componente C está falho, ainda será possível verificar as discrepâncias geradas pelo conjunto de contribuintes CO_j. Dessa forma, será necessário supor que algum componente $C' \in CO_j$ está falho. Mas como $CO_j \subset CO_i$, então uma hipótese de falha, H_0, que contém $\{C, C'\}$ não será minimal, pois haverá alguma outra hipótese de falha, H_1, que contém C' mas não C, tal que $H_1 \subset H_0$.

A Figura 3.4 mostra essa relação entre os sintomas, conjuntos contribuintes e hipóteses de falha. Nessa figura, são representados três sintomas (X, Y e Z) e os respectivos conjuntos de contribuintes. As hipóteses de falha são os conjuntos de componentes minimais e que contém ao menos um componente de cada conjunto de contribuinte (conjuntos de corte minimais).

Na próxima seção, apresentamos o Algoritmo de Reiter, que encontra todos os conjuntos de corte minimais de uma família de conjuntos quaisquer, e pode ser usado para encontrar os diagnósticos de um sistema a partir dos conjuntos de contribuintes.

3.5 Algoritmo de Reiter: *Minimal Hitting Set*

O Algoritmo de Reiter [36], também chamado de *Minimal Hitting Set*, é um algoritmo geral que permite encontrar todos os conjuntos de corte minimais de uma família de conjuntos quaisquer.

Seja F uma família de conjuntos quaisquer. Um *grafo de corte* é um grafo orientado acíclico

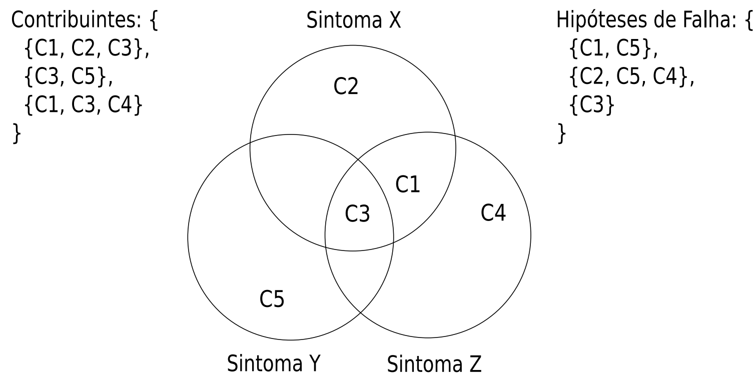


Figura 3.4: Relação entre sintomas, conjunto de contribuintes e hipóteses de falha.

e enraizado no qual cada nó n é rotulado por um conjunto $S \in F$, e os arcos saindo de n são rotulados por elementos de S. Para todo nó n, $H(n)$ é o conjunto formado pelos rótulos dos arcos no caminho da raiz até n. Considere $H(n) = \emptyset$, quando n é o nó raiz. O grafo é gerado e expandido em largura e sempre que for necessário obter o rótulo para um novo nó, n', é procurado um novo conjunto $S' \in F$ tal que $S' \cap H(n') = \emptyset$. Caso não exista tal conjunto em F, o nó é rotulado com @. Após gerar e expandir o grafo inteiro, todo nó n rotulado por @ tem em $H(n)$ um conjunto de corte para F. Um grafo de corte não contém todos os possíveis conjuntos de corte para F, mas contém todos os conjuntos de corte minimais.

Exemplo 3.3 - Grafo de corte. Seja $F = \{\{2,4\},\{4,1\},\{2,3\},\{3,5\},\{4\}\}$ uma família de conjuntos para o qual deseja-se encontrar os conjuntos de corte minimais. A Figura 3.5 mostra o grafo de corte gerado para F. Os nós rotulados por @ são conjuntos de corte.

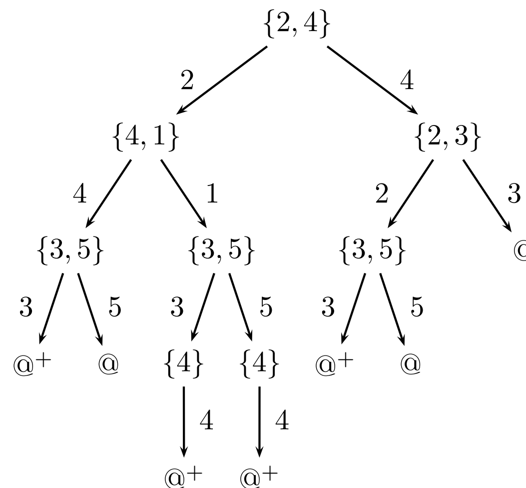


Figura 3.5: Grafo de corte para a família de conjuntos $F = \{\{2,4\},\{4,1\},\{2,3\},\{3,5\},\{4\}\}$. Todo nó n com rótulo @ define em $H(n)$ um conjunto de corte minimal para F, enquanto os nós rotulados por $@^+$ também são conjuntos de corte mas não são minimais.

□

No Exemplo 3.3, para cada nó criado, a família de conjuntos F é acessada para obter o rótulo desse novo nó. Assim, F é acessada ao todo 14 vezes. Algumas estratégias de poda e otimização podem ser usadas para reduzir o tamanho desse grafo e também minimizar o número de vezes que F é acessado. O Algoritmo de Reiter explora essas estratégias, sendo capaz de gerar um grafo no qual todos os conjuntos de corte (definidos pelos nós rotulados por @) são minimais. A seguir descrevemos o Algoritmo de Reiter (Algoritmo 1) em detalhes.

Seja F uma família de conjuntos quaisquer. Se F é vazio, o algoritmo termina devolvendo um conjunto vazio, informando que não foi encontrado nenhum conjunto de corte para F. Suponha que F não seja vazio. No início do algoritmo é escolhido um conjunto qualquer de F para rotular a raiz do grafo. O algoritmo segue expandindo o grafo em largura, enquanto houver um nó que ainda possa ser expandido. Seja n o nó a ser expandido e S o conjunto de F que rotula n. Para cada $s \in S$ é criado um arco saindo de n e rotulado por s. O algoritmo seleciona um desses arcos, s_arco rotulado por s, e verifica qual dos quatro passos a seguir é aplicável para obter o nó que s_arco deve apontar:

- Passo **P1**, se existe no grafo um nó n', tal que $H(n') = H(n) \cup \{s\}$, s_arco deve apontar parar n'. Esse passo permite reaproveitar a informação já presente no grafo ao invés de obter um novo conjunto de F, como mostraremos no passo **P4**.

- Passo **P2**, se existe no grafo um nó n', tal que $H(n')$ seja um subconjunto próprio de $H(n) \cup \{s\}$, o s_arco deve apontar para um nó que finaliza esse ramo do grafo, chamado de *nó fechado* e indicado por um "X". Esse passo evita que sejam expandidos nós de caminhos que levariam a conjuntos de corte não minimais.

- Passo **P3**: se os passos anteriores não são aplicáveis, o algoritmo verifica se existe algum nó n' no grafo, com rótulo S', tal que $(H(n) \cup \{s\}) \cap S' = \emptyset$. Se existir tal nó, um novo nó é criado e rotulado pelo conjunto S'. Esse passo também permite reaproveitar informações contidas no grafo, antes de obter um novo conjunto de F, como mostrado no passo a seguir.

- Passo **P4**: Caso o passo **P3** não seja aplicável, um conjunto $S' \in F$ que satisfaça a restrição $S' \cap (H(n) \cup \{s\}) = \emptyset$ deve ser obtido. Se não for possível encontrar um S' que satisfaça a restrição, o s_arco deverá apontar para um novo nó, n_1, rotulado por @, indicando que $H(n_1)$ é um conjunto de corte para F. Caso seja encontrado um S' que satisfaça a restrição, um novo nó, n_1, é criado e rotulado por S'. Em seguida, o algoritmo verifica se o passo $P4'$ é aplicável.
  - Passo **P4'**: Se existe no grafo um nó n_0 rotulado por S_0, tal que S' é um subconjunto próprio de S_0, o rótulo de n_0 deve ser renomeado para S' e todos os arcos saindo de n_0 com rótulos pertencentes ao conjunto resultante de $S_1 - S'$, bem como as respectivas subárvores, devem ser eliminadas. Esse passo permite remover partes do grafo que

permitiriam encontrar conjuntos de corte minimais antes de S' fazer parte do grafo, mas considerando S' isso não é mais possível.

Ao final do algoritmo, para todo nó n rotulado por @, $H(n)$ é um conjunto de corte minimal para F. O Algoritmo 1 descreve o funcionamento do Algoritmo de Reiter, considerando as correções apresentadas em [23].

Algoritmo 1: MinimalHittingSet(F)

Entrada: F: uma família de conjuntos.
Saída: todos os conjuntos de corte minimais de F.

1 Escolha um conjunto de F para nomear a raiz (nível 0);
2 **para cada** *nó n no nível i* **faça**
3   **se** *n tem um rótulo S* **então**
4   **para cada** $s \in S$ **faça** crie um *s_arco* saindo de n com rótulo s.
5     $H(n) \leftarrow$ conjunto de rótulos do caminho da raiz até n.
    ▷ `passo P1`
6   **se** *existe algum nó n' tal que* $H(n') = H(n) \cup \{s\}$ **então**
7   faça o *s_arco* de n apontar para n'.
    ▷ `passo P2`
8   **senão se** *existe n' com rótulo @ tal que* $H(n') \subset (H(n) \cup \{s\})$ **então**
9   feche o *s_arco*.
    ▷ `passo P3`
10   **senão se** *existe n' rotulado como S' tal que* $S' \cap (H(n) \cup \{s\}) = \emptyset$ **então**
11   faça o *s_arco* de n apontar para um novo nó rotulado por S'.
    ▷ `passo P4`
12   **senão**
13   faça o *s_arco* apontar para um novo nó m e rotule m pelo primeiro elemento S' de F tal que $S' \cap H(m) = \emptyset$. Se tal conjunto não existir, então rotule m como @.
        ▷ `passo P4'`
14   **se** *existe algum nó n' com um rótulo S_1 tal que* $S' \subset S_1$ **então**
15   renomeie o nó n' como S' e remova todos os arcos saindo de n' que foram nomeados pelos elementos de $S_1 - S'$.
16 **fim**
17 Repita o passo 2 para $i + 1$.
18 **devolva** $\{H(n) \mid n \text{ é um nó rotulado com } @\}$

Exemplo 3.4 - Algoritmo de Reiter. Seja $F = \{\{2,4\},\{4,1\},\{2,3\},\{3,5\},\{4\}\}$. A Figura 3.6 mostra como o grafo vai sendo alterado durante a execução do Algoritmo 1. Nessa figura, cada nó possui um índice do lado direito usado como identificador. Um nó com índice i será referenciado como n_i.

No início do algoritmo é selecionado um conjunto qualquer de F para rotular a raiz do grafo, no caso, o conjunto $\{2,4\}$ (Figura 3.6a). A partir da raiz são criados dois arcos rotulados por 2 e 4. Suponha que *s_arco* seja o arco rotulado por 2. Como não existe nenhum nó n' no grafo que tenha $H(n') = \{2\}$, o passo $P1$ não é aplicável. Também não existe nenhum nó rotulado por @, então os passo $P2$ também não é aplicável. O passo $P3$ não é aplicável pois não existe nenhum nó n' no grafo tal que $H(n') \cap \{2\} = \emptyset$. Assim, o passo $P4$ é aplicado, gerando um novo nó rotulado

por $\{4, 1\}$. O passo $P4'$ não é aplicável pois não existe no grafo um nó n', rotulado por S_1 tal que $\{4, 1\} \subset S_1$. Seja *s_arco* o outro arco saindo da raiz do grafo, rotulado por 4. Nesse caso, os passos $P1$ até $P3$ não são aplicáveis, de maneira análoga ao que foi feito para o arco rotulado por 2, e ao executar o passo $P4$ é criado um novo nó com rótulo $\{2, 3\}$. Novamente o passo $P4'$ não é aplicável nessa situação (Figura 3.6b).

O algoritmo gera os arcos saindo do nó n_2, rotulados por 4 e 1. Para o arco rotulado por 4 não é possível executar os passos de $P1$ até $P3$, e ao executar o passo $P4$ é gerado um novo nó, rotulado por $\{3, 5\}$. Novamente o passo $P4'$ não é aplicável. Para o arco rotulado por 1, os passos $P1$ e $P2$ não são aplicáveis, mas para n_4 vale que $H(n_4) \cap H(n_2) \cup \{1\} = \emptyset$, então o passo $P3$ é aplicável e o rótulo $\{3, 5\}$ é usado para rotular um novo nó n_4 (Figura 3.6c).

O algoritmo expande o nó n_3, executando o passo $P2$, que faz com que o arco rotulado por 2 aponte para o nó n_4; para o arco rotulado por 3, é executado o passo $P4$ e o algoritmo cria um novo nó rotulado por @ (Figura 3.6d). O nó n_4 é expandido. Para o arco rotulado por 3, o algoritmo executa o passo $P2$ e fecha esse arco. Para o arco rotulado por 5, é executado o passo $P4$, gerando um nó rotulado por @ (Figura 3.6e).

O nó n_5 é expandido. Para o arco rotulado por 3 é executado o passo $P4$, fazendo com que esse arco aponte para um novo nó rotulado por 4. Nesse caso, o passo $P4'$ é aplicado, trocando o rótulo de n_1 para $\{4\}$ e removendo o arco rotulado por 2 e toda a subárvore (Figura 3.6f). Como não é possível expandir mais nenhum nó, o algoritmo termina e devolve os conjuntos de corte $\{4, 2, 3\}$ e $\{4, 3\}$.

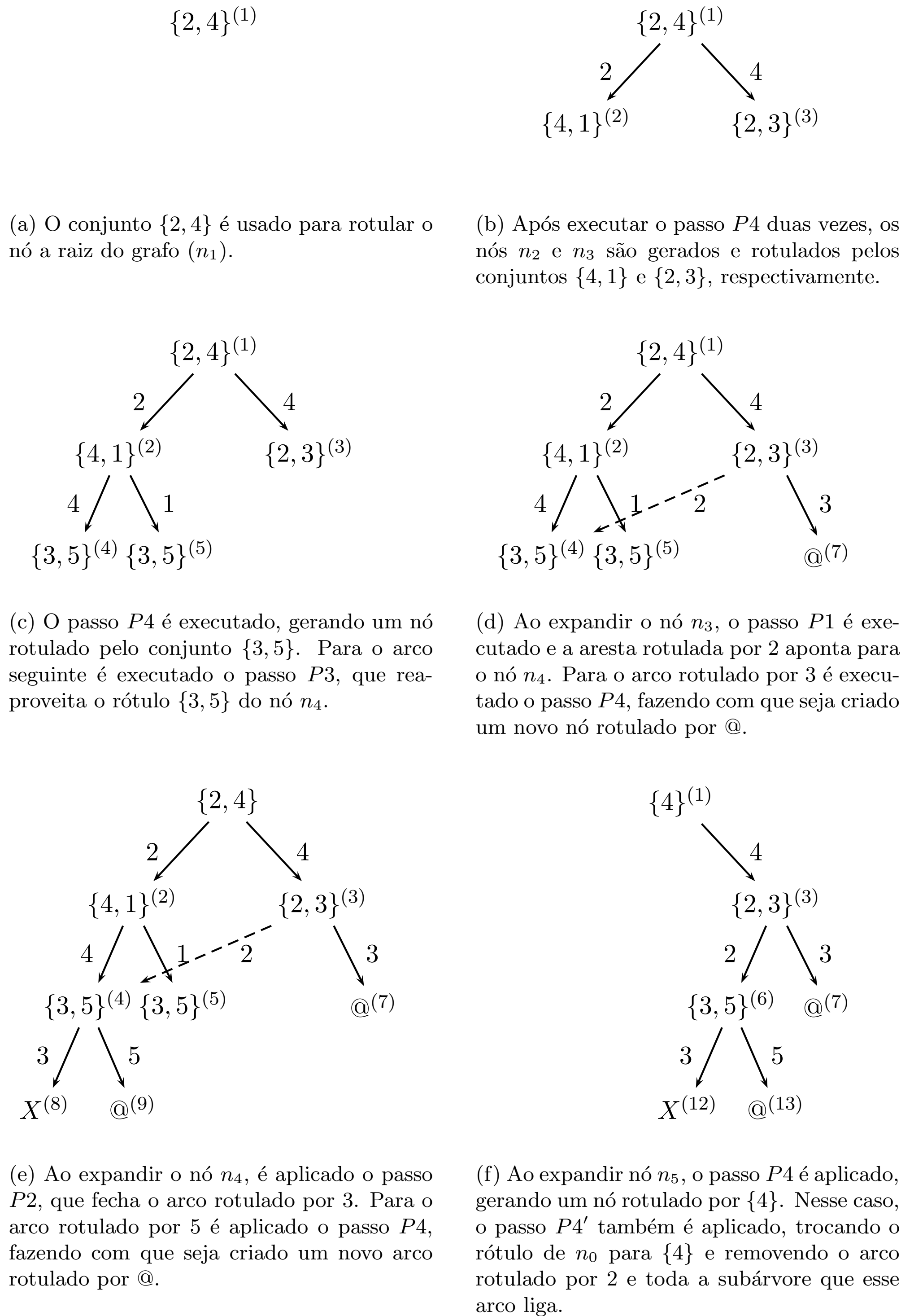
□

É importante observar que utilizando o Algoritmo 1, um novo elemento de F é visitado somente quando o passo $P4$ é executado. No caso do Exemplo 3.4, o conjunto F foi acessado 7 vezes, enquanto usando a árvore de busca apresentada no Exemplo 3.3, o conjunto F foi acessado 14. Assim, é possível observar um ganho de desempenho utilizando o Algoritmo 1.

3.6 MBD com o algoritmo de Reiter

Segundo o Teorema 3.1, é possível obter todos os diagnósticos possíveis para um sistema encontrando os conjuntos de corte minimais para os conjuntos de contribuintes desse sistema. O que normalmente ocorre é que o conjunto de contribuintes não está previamente disponível, sendo necessário um mecanismo que possibilite a geração de cada conjunto de contribuintes de uma forma metódica.

Pela Definição 3.5, se $CO \subseteq COMP$ é um conjunto de contribuintes para o sistema $\langle SD, COMP, OBS \rangle$, então $SD \cup OBS \cup \{ok(C) | C \in CO\}$ é inconsistente, e consequentemente, $SD \cup OBS \cup \{ok(C) | C \in COMP\}$ também é inconsistente. Com um provador de teoremas completo e correto é possível en-



(a) O conjunto $\{2, 4\}$ é usado para rotular o nó a raiz do grafo (n_1).

(b) Após executar o passo $P4$ duas vezes, os nós n_2 e n_3 são gerados e rotulados pelos conjuntos $\{4, 1\}$ e $\{2, 3\}$, respectivamente.

(c) O passo $P4$ é executado, gerando um nó rotulado pelo conjunto $\{3, 5\}$. Para o arco seguinte é executado o passo $P3$, que reaproveita o rótulo $\{3, 5\}$ do nó n_4.

(d) Ao expandir o nó n_3, o passo $P1$ é executado e a aresta rotulada por 2 aponta para o nó n_4. Para o arco rotulado por 3 é executado o passo $P4$, fazendo com que seja criado um novo nó rotulado por @.

(e) Ao expandir o nó n_4, é aplicado o passo $P2$, que fecha o arco rotulado por 3. Para o arco rotulado por 5 é aplicado o passo $P4$, fazendo com que seja criado um novo arco rotulado por @.

(f) Ao expandir nó n_5, o passo $P4$ é aplicado, gerando um nó rotulado por $\{4\}$. Nesse caso, o passo $P4'$ também é aplicado, trocando o rótulo de n_0 para $\{4\}$ e removendo o arco rotulado por 2 e toda a subárvore que esse arco liga.

Figura 3.6: Sequência de passos executados pelo Algoritmo 1 para encontrar todos os conjuntos de corte minimais de $F = \{\{2, 4\}, \{4, 1\}, \{2, 3\}, \{3, 5\}, \{4\}\}$. Na figura, cada nó tem em seu canto superior direito um índice utilizado para identificá-los. Um nó com índice i é referenciado por n_i

contrar os conjuntos de contribuintes verificando as árvores de prova usadas pelo provador de teore-

mas para verificar a inconsistência de $SD \cup OBS \cup \{ok(C)|C \in COMP\}$[5]. Para cada árvore de prova são verificados quais os fatos a respeito do funcionamento dos componentes (sentenças com o predicado unário *ok* e com a variável substituída por uma constante) fazem parte da prova, sendo o componente em cada um desses fatos usado para compor um conjunto de contribuintes. Suponha que para construir uma árvore de prova foram usados os seguintes fatos: $\{ok(C_1),\ ok(C_2), \ldots, ok(C_k)\}$, sendo $C_i \in COMP$, o conjunto formado pelos componentes $\{C_1, C_2, \ldots, C_k\}$ é um conjunto de contribuintes.

Exemplo 3.5 - Árvore de prova. Seja $P_{MBD} = \langle SD, COMP, OBS \rangle$ um problema de diagnóstico, com $COMP = \{C_1, C_2\}$, $OBS = \{P, \neg R\}$, e SD definido da seguinte forma[6]:

$$\neg ok(C_1) \vee P \vee Q$$
$$\neg ok(C_1) \vee \neg Q \vee \neg P$$
$$\neg ok(C_2) \vee Q \vee R$$
$$\neg ok(C_2) \vee \neg Q \vee \neg R$$

Uma árvore de prova que mostra a inconsistência de $SD \cup OBS \cup \{ok(C)|C \in COMP\}$ é ilustrada na Figura 3.5. Entre as sentenças usadas nessa árvore de prova estão: $ok(C_1)$ e $ok(C_2)$, formando o conjunto de contribuintes: $\{C_1, C_2\}$.

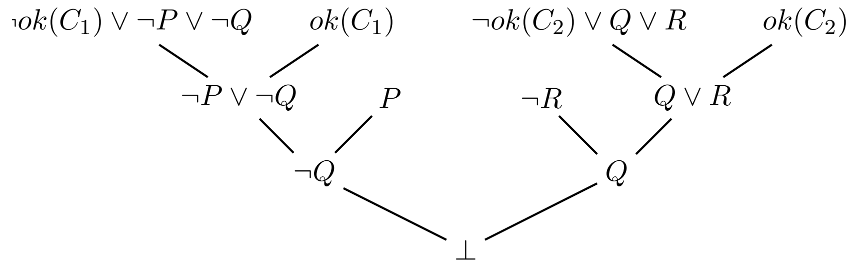


Figura 3.7: Exemplo de uma árvore de prova baseada em resolução, utilizada para derivar a inconsistência em $SD \cup OBS \cup \{ok(C)|C \in COMP\}$.

□

Um problema dessa abordagem é que o provador de teoremas pode gerar diversas árvores de prova distintas para um mesmo conjunto de contribuintes, fazendo com que esse processo de busca (de todos os conjuntos de contribuintes) consuma mais tempo do que o necessário. Como mostraremos a seguir, o Algoritmo 1 pode ser usado para controlar a busca dos conjuntos de contribuintes.

[5]Cada árvore de prova pode ser construída por sucessivas aplicações do método de inferência de *resolução*, até que seja derivada uma cláusula contendo apenas o símbolo falso (⊥)

[6]As sentenças desse modelo foram convertidas em um conjunto de cláusulas (i.e., disjunções de literais em que todas as variáveis são universais). Para obter essas cláusulas, as sentenças descrevendo o modelo foram convertidas para a forma normal conjuntiva (i.e., uma conjunção de cláusulas) e as condições foram eliminadas.

Apesar do Algoritmo 1 usar uma família de conjuntos F previamente definida, note que F somente é acessado no início do algoritmo (Linha 1), para obter o conjunto usado para rotular a raiz do grafo, e no passo $P4$, para obter um conjunto $S' \in F$, tal que $S' \cap (H(n) \cup \{s\}) = \emptyset$, usado para rotular um novo nó do grafo. Dessa forma, se uma família de conjuntos tem uma determinada propriedade (por exemplo, cada elemento dessa família é um conjunto de contribuintes) e existe uma função TP que satisfaça as restrições do passo $P4$, capaz de devolver os conjuntos dessa família, o Algoritmo 1 pode ser modificado para encontrar todos os conjuntos de corte minimais dessa família de conjuntos. O uso da função TP para obter os conjuntos usados para rotular os nós do grafo permite que a família de conjuntos F seja definida de forma implícita, sendo gerada conforme as chamadas a TP são feitas.

Seja $P_{MBD} = \langle SD, COMP, OBS \rangle$ um problema de diagnóstico e TP uma função que recebe como parâmetros: o modelo do sistema, SD um conjunto de componentes $okCOMPS \subseteq COMP$ e as observações, OBS; e devolve um conjunto de contribuintes, caso exista, obtido a partir da árvore de prova usada para verificar a inconsistência da fórmula:

$$SD \cup OBS \cup \{ok(C) | C \in okCOMPS\} \quad (3.5)$$

Caso a função TP não encontre nenhum conjunto de contribuintes, deverá ser devolvido um conjunto vazio. Note que, dado um conjunto de componentes $CMP \subseteq COMP$, a chamada $TP(SD, COMP - CMP, OBS)$ devolve um conjunto de contribuintes CO tal que $CO \cap CMP = \emptyset$. Assim, a função TP, implementada de acordo com as características apresentadas, pode ser utilizada no Algoritmo 1 para fazer a busca dos conjuntos de contribuintes de um sistema $\langle SD, COMP, OBS \rangle$. O Algoritmo 2 mostra os pontos do Algoritmo 1 nos quais foram feitas as alterações para usar a função TP citada.

Algoritmo 2: MinimalHittingSet-MBD($\langle SD, COMP, OBS \rangle$, TP)

Entrada: $\langle SD, COMP, OBS \rangle$: um problema de diagnóstico,
TP: uma função definida como $TP(SD, okCOMPS, OBS)$ que devolve um conjunto de contribuintes verificando a inconsistência da Fórmula 3.5, ou $\emptyset$ caso não seja encontrado nenhum conjunto de contribuintes.
Saída: todas as hipóteses de falhas em $\langle SD, COMP, OBS \rangle$.

1 Chame $TP(SD, COMP, OBS)$ e use o conjunto de contribuintes devolvido para rotular a raiz (nível 0) ;
2 **para cada** *nó n no nível i* **faça**
...
▷ passo P4
12 **senão**
13 faça o *s_arco* apontar para um novo nó m e rotule m pelo conjunto de contribuintes S' obtido na chamada da função $TP(SD, COMP - (H(n) \cup \{s\}), OBS)$. Se a função devolver $\emptyset$, rotule m como @ ;
▷ passo P4'
...
...
fim

O Algoritmo 2 recebe como entrada um problema de diagnóstico e a referência para a função TP usada para gerar os conjuntos de contribuintes. No início do algoritmo (Linha 1) é feita a chamada para TP, que devolve um conjunto de contribuintes usado para rotular a raiz do grafo. Sempre que o passo $P4$ for aplicável, a função TP é chamada (Linha 13) para obter um conjunto de contribuintes. Note que os componentes passados para TP nesse passo são $COMP - (H(n) \cup \{s\})$, o que faz com que não sejam usados os fatos a respeito do comportamento correto dos componentes em $H(n) \cup \{s\}$. Essa técnica é conhecida como *suspensão de restrições* [10], que consiste em não fazer nenhuma suposição a respeito do comportamento correto ou falho de um componente C. Dessa forma, os axiomas de simulação e de satisfação de restrições não podem ser usados para inferir valores para novas variáveis. Além disso, o componente C, representado por uma constante, não pode ser usado com o predicado *ok* em nenhuma árvore de prova usada pelo provador de teoremas para derivar a inconsistência na fórmula:

$$SD \cup OBS \cup \{ok(C)|C \in (H(n) \cup \{s\})\}$$

Assim, mesmo havendo em SD uma sentença descrevendo o comportamento correto de C (na forma $ok(C) \rightarrow \ldots$), não será possível usar essa sentença para derivar um falso na árvore de prova, pois não haverá um fato $ok(C)$ para o qual o método de resolução possa ser aplicado.

Observe que o Algoritmo 1 minimiza o número de vezes que o passo $P4$ é executado. O mesmo acontece com o Algoritmo 2, minimizando o número de vezes que o provador de teoremas é chamado, que é uma operação com alto custo computacional.

Apesar da função TP citada usar um provador de teoremas na busca dos conjuntos de contribuintes, também é possível usar um mecanismo de propagação de restrições [10, 14]. Nesse caso, o comportamento de cada componente é descrito por um conjunto de restrições com variáveis (entradas e saídas do componente), a partir das quais é possível inferir o valor de outras variáveis. As observações feitas no sistema fornecem valores para algumas variáveis. Durante o processo de propagação, se for inferido o valor para uma variável que é diferente do valor observado, os componentes que têm seus comportamentos definidos por restrições usadas na propagação formam um conjunto de contribuintes (equivalente às inconsistências encontradas pelo provador de teoremas). Apesar da propagação de restrições permitir um controle melhor sobre o processo de inferência, existem dois pontos que devem analisados: (1) o processo de propagação deve ser completo, ou seja, todas as possíveis combinações devem ser geradas; (2) a execução de várias chamadas para o propagador pode fazer com que diversas inferências sejam refeitas, levando a uma degradação no desempenho. Com o uso de um sistema de manutenção da verdade (ATMS) [11] é possível evitar que inferências sejam refeitas. Uma das funções do ATMS é armazenar as inferências feitas por um propagador (ou um provador de teoremas) junto das suposições que o permitiram fazer essas inferências. No caso do diagnóstico, as suposições dizem respeito ao funcionamento correto ou falho dos componentes. Antes de fazer uma inferência o propagador usa as suas suposições para consultar

o ATMS, que devolve o resultado da inferência, caso ele tenha essa informação armazenada, e evita que o propagador faça as inferências. Se o ATMS não tiver a informação, o propagador faz as inferências e passa o resultado para ser armazenado no ATMS. A abordagem usada em [14] mostra com mais detalhes como é feita a integração de um ATMS em um sistema de diagnóstico.

3.7 Exemplo de diagnóstico com MBD

Nesta seção é apresentado como o Algoritmo de Reiter pode ser usado para encontrar as falhas do circuito do Exemplo 3.1, que tem seu modelo e as observações descritas no Exemplo 3.2.

3.7.1 Exemplo de Diagnóstico com o Algoritmo de Reiter

As observações feitas no circuito do Exemplo 3.1 indicam o sintoma: "A saída esperada em F é 12, mas foi observado 10". O Algoritmo de Reiter é executado e como resultado da primeira chamada ao provador de teoremas é encontrado o conjunto de contribuintes $\{M_1, M_2, A_1\}$, que rotula a raiz do grafo (Figura 3.8a). Cada elemento desse conjunto é usado para rotular um arco saindo da raiz. O algoritmo começa a expandir o grafo, buscando um nó para o qual o arco rotulado por M_1 deve apontar (passos P1 a P3). Como não há no grafo nós que possam ser reutilizados, o provador de teoremas é chamado (passo P4), considerando a suspensão de restrições para o componente M_1. O provador de teoremas não encontra inconsistência e um novo nó é criado e rotulado por "@". O mesmo acontece para o arco rotulado como A_1. No caso do arco rotulado como M_2, também não há nós no grafo para os quais esse arco possa apontar. O provador de teoremas é novamente chamado e o conjunto de contribuintes $\{M_1, M_3, A_1, A_2\}$ gerado é usado para rotular um novo nó. O grafo resultante após esses três passos é mostrado na Figura 3.8b. O algoritmo passa para o próximo nível do grafo e expande o nó rotulado por $\{M_1, M_3, A_1, A_2\}$. Partindo desse nó, os arcos rotulados por M_1 e A_1, são fechados, pois não é possível chegar a um conjunto minimal seguindo por eles (passo P2). No caso do arco rotulado como M_3, nenhum nó pode ser reaproveitado, então é chamado o provador de teoremas considerando a suspensão de restrições dos componentes M_2 e M_3. O provador não encontra inconsistência e um novo nó é criado e rotulado por "@". O mesmo acontece para o arco rotulado como A_2. Como não é possível continuar expandindo o grafo, o algoritmo termina e devolve todos os conjuntos $H(n)$, no qual n é rotulado por "@" (Figura 3.8c). Nesse caso, as hipóteses de falha são: $\Delta = \{\{M_1\}, \{A_1\}, \{M_2, M_3\}, \{M_2, A_2\}\}$.

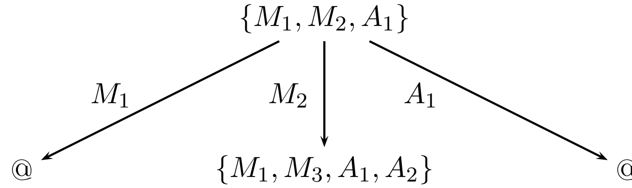
Na próxima seção, mostraremos como é possível reduzir o conjunto de hipóteses de falha encontradas no circuito do Exemplo 3.1, através da discriminação de hipóteses.

3.7.2 Discriminação de Hipóteses

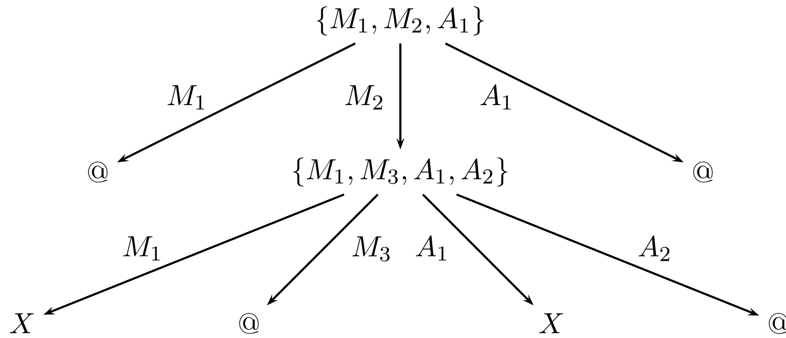
Na seção anterior, foram encontradas as seguintes hipóteses de falha para o circuito do Exemplo 3.1: $\Delta = \{\{M_1\}, \{A_1\}, \{M_2, M_3\}, \{M_2, A_2\}\}$. Seja $P^0_{MBD} = \langle SD, COMP, OBS \rangle$ o problema de diagnóstico para esse circuito, conforme descrito no Exemplo 3.2.

$\{M_1, M_2, A_1\}$

(a) Rotulando a raiz do grafo com os contribuintes $\{M_1, M_2, A_1\}$.



(b) Conjuntos de corte minimais reconhecidos e o novo contribuinte $\{M_1, M_3, A_1, A_2\}$ adicionado (passo **P4**).



(c) Grafo ao final da execução do algoritmo.

Figura 3.8: Grafo gerado pelo Algoritmo de Reiter durante o processo de diagnóstico do circuito da Figura 3.2.

Suponha que seja medida a saída do componente M_1, obtendo-se o valor 6. Essa observação, representada por: $out_1(M_1) = 6$, é adicionada às observações já existentes, OBS, gerando um novo conjunto de observações, OBS_1. Dessa forma, pode-se definir um novo problema de diagnóstico: $P^1_{MBD} = \langle SD, COMP, OBS_1 \rangle$, no qual são obtidas as seguintes hipóteses de falha (através do mesmo procedimento exemplificado na Seção 3.7.1):

$$\Delta_1 = \{\{A_1\}, \{M_2, M_3\}, \{M_2, A_2\}\}$$

Na sequência, é feita a medição na saída do componente M_2, na qual obtém-se o valor 4, representado por: $out_1(M_2) = 4$. Novamente, essa observação é adicionada às observações em OBS_1, gerando o conjunto de observações OBS_2. Dessa vez, a solução para o problema de diagnóstico $P^2_{MBD} = \langle SD, COMP, OBS_2 \rangle$ é:

$$\Delta_2 = \{\{M_2, M_3\}, \{M_2, A_2\}\}$$

Por fim, é feita a medição na saída do componente M_3, na qual obtém-se o valor 6, representada

pela observação: $out_1(M_3) = 6$. Essa nova observação é adicionada ao conjunto de observações OBS_2, gerando o conjunto de observações OBS_3. A solução para o problema de diagnóstico $P^3_{MBD} = \langle SD, COMP, OBS_3 \rangle$ é:

$$\Delta_3 = \{\{M_2, A_2\}\}$$

Nesse caso, Δ_3 contém somente uma hipótese de falha, indicando que esses dois componentes (M_2 e A_2) são os componentes realmente falhos no sistema.

3.8 Complexidade do MBD

Encontrar todas as hipóteses de falha em um sistema $\langle SD, COMP, OBS \rangle$ pode ser visto como reconhecer os componentes pertencentes a cada um dos possíveis Δ, tal que:

$$SD \cup OBS \cup \{ok(C) | C \in COMP - \Delta\} \cup \{\neg ok(C) | C \in \Delta\}$$

seja consistente (Definição 3.4). Uma abordagem simples para encontrar Δ, consiste em enumerar todos os subconjuntos de $COMP$ e supor que os componentes desse subconjunto estão falhos. Isso pode ser feito substituindo o Δ utilizado na Definição 3.4 por cada um desses subconjuntos e verificando a consistência da fórmula. Essa abordagem consome tempo $O(2^{|COMP|})$, para enumerar dos os subconjuntos de $COMP$. Para cada subconjunto, o provador de teoremas é chamado para verificar a consistência da fórmula na Definição 3.4. Suponha que essa chamada consuma tempo $O(f(|SD|, |COMP|))$ no pior caso. Assim, o consumo de tempo total para fazer o diagnóstico, no pior caso, é dado por $O(2^{|COMP|} \times f(|COMP|))$, que é exponencial. Com o Algoritmo de Reiter é possível obter melhorias de desempenho, mas resultados apresentados em [42] mostram que encontrar todos os conjuntos de corte minimais de uma família de conjuntos é NP-difícil.

É possível encontrar na literatura diferentes abordagens para mostrar que o problema de diagnóstico é NP-difícil. Em [19], o problema de diagnóstico é analisado como um problema de consistência em lógica considerando somente cláusulas de Horn. O resultado mostra que: encontrar a segunda hipótese de falha torna o problema NP-difícil; se a descrição do sistema contém um modelo de falhas, para encontrar a primeira hipótese de falha já é um problema NP-difícil. A abordagem apresentada em [41] trata o problema de diagnóstico dedutivo como uma instância de um problema de abdução e também mostra que o problema é NP-difícil, independente da estrutura utilizada na representação do problema.

3.9 Considerações finais

Nessa seção, apresentamos os principais conceitos sobre Diagnóstico baseado em Modelos, uma técnica utilizada para encontrar componentes falhas em sistemas físicos. Além disso, também

apresentamos um importante algoritmo utilizado para fazer o diagnóstico: o *Minimal Hitting Set*, também conhecido como algoritmo de Reiter. Também foi apresentada uma ideia intuitiva sobre a complexidade do problema de diagnóstico.

No próximo capítulo, apresentaremos a técnica de diagnóstico hierárquico baseado em modelos, que permite fazer o diagnóstico utilizando múltiplos níveis de abstrações, que permite obter ganhos em termos de esforço computacional, em relação ao diagnostico baseado em modelos tradicional.

Capítulo 4

Diagnóstico Hierárquico Baseado em Modelos

Uma das limitações da técnica de diagnóstico baseado em modelos é o custo computacional necessário para se encontrar todas as hipóteses de falha em um sistema, para depois discriminá-las. O uso de abstrações é uma forma de, na prática, tratar esse problema com maior eficiência [21, 39, 31]. Abstrações são utilizadas, mesmo que intuitivamente, ao se modelar um sistema. Durante a construção do modelo, certas decisões são tomadas a respeito de quais características do sistema o modelo deve capturar, isto é, o nível de abstração adotado para representar esse sistema. Por exemplo, em um circuito digital os componentes não trabalham exatamente com os valores 0 e 1, mas sim com valores analógicos que são interpretados como valores digitais. Apesar dessa característica estar presente no circuito físico, um modelo pode capturar somente uma visão mais abstrata do sistema e considerar que os sinais no circuito são 0 e 1. Certamente, o nível de abstração considerado para a construção do modelo também determinará o nível de abstração do diagnóstico obtido.

Durante o processo de diagnóstico, as abstrações são úteis para permitir que grandes partes do sistema, consideradas corretas na presença dos sintomas, sejam isoladas durante a busca pelos componentes falhos [21]. Por exemplo, quando um técnico percebe um comportamento anormal em um carro, a abordagem inicial consiste em isolar um ou mais subsistemas desse carro (por exemplo, parte mecânica, parte elétrica, sistema de freios, sistema de ignição, etc.) envolvidos no problema. Nessa abordagem, o técnico não precisa necessariamente conhecer os detalhes a respeito de todos os componentes de todos os subsistemas do carro, mas somente ter conhecimento suficiente para descartar a possibilidade de haver falha em algumas partes. Uma vez que o subsistema com problema foi identificado, esse passa a ser o foco do diagnóstico, reduzindo consideravelmente o número de componentes que devem ser verificados. Esse processo de isolar as partes abstratas que compõem o sistema e depois executar o mesmo processo nas partes internas, pode ser feito sucessivamente até que sejam encontrados os componente realmente falhos.

O *diagnóstico hierárquico baseado em modelos* (*Hierarchical Model Based Diagnosis* - HMBD) [21, 31, 39, 2, 8, 18] é uma técnica que permite encontrar componentes falhos em um sistema descrito por diversos níveis de abstração. Cada nível forma uma visão mais mais abstrata do sistema,

definindo um conjunto de abstrações em relação a outro nível. A abordagem para fazer o diagnóstico considerando hierarquias consiste em isolar os componentes falhos em um nível mais abstrato e usar essa informação para guiar o processo de raciocínio nos níveis mais detalhados [31,8]. Dessa forma, para encontrar um diagnóstico no nível de maior detalhe, não é necessário considerar a falha de todos os componentes do modelo durante a geração de hipóteses, permitindo assim obter ganhos de desempenho.

Nesse capítulo, consideramos que os níveis de abstração descrevendo um sistema serão fornecidos, não importando a forma como foram construídos. Por questões de terminologia, sempre que houver necessidade de se referenciar às técnicas de diagnóstico baseado em modelos, como apresentada no Capítulo 3, que usa somente o modelo menos abstrato do sistema para fazer o diagnóstico, serão utilizados os termos *MBD tradicional* e *abordagem tradicional* quando o contexto permitir.

Como existem diversas formas distintas de tratar um problema de diagnóstico considerando hierarquias [21, 31, 39, 2, 8, 18], não é possível encontrar uma formalização única que caracterize esse problema. Dessa forma, optamos por introduzir nesse capítulo as ideias básicas de diagnóstico hierárquico, a forma de representação do problema, as técnicas e os algoritmos, com base nos trabalhos de Mozetič [31] e Chittaro & Ranon [8].

4.1 Conceitos básicos

4.1.1 Modelo Abstrato

Informalmente, o termo "abstração" pode ser definido como o processo de mapear uma representação de um problema em uma nova representação mais simples, ou ainda, como o processo de considerar aquilo que é importante em um problema sem se preocupar com detalhes irrelevantes [22]. As abstrações que são normalmente utilizadas para fazer o diagnóstico hierárquico são:

- *Abstrações Comportamentais* [31], que redefinem o comportamento do sistema de uma forma simplificada, preservando certas propriedades importantes do seu comportamento original. As abstrações comportamentais, normalmente, causam mudanças no domínio no qual o sistema opera, por exemplo: um componente digital interpreta suas entradas e saídas como valores 0 e 1, mas internamente a esse componente pode haver outros componentes que operam com valores analógicos.

- *Abstrações Estruturais* [21, 8], que são construídas a partir de agregações de componentes, gerando outros componentes (*caixa preta*), e permitem descrever o sistema em vários níveis de abstração estrutural, por exemplo: o motor de um carro é composto de cabeçote, válvulas, pistões, etc. O componente gerado a partir de uma abstração estrutural é chamado de *componente abstrato*.

Nesse trabalho, serão consideradas somente abstrações do tipo estrutural.

4.1.2 Modelos com Múltiplos níveis de abstração

Nível de abstração, ou simplesmente *nível*, é cada um dos níveis utilizados para representar um sistema. Para cada nível i representando o sistema é associado um modelo, SD_i, que descreve o sistema. O modelo que não tem nenhuma abstração, representando o sistema em todos os seus detalhes, SD_0, é chamado de *modelo base*. Um nível $i+1$ define um conjunto de abstrações formadas a partir do nível i, sendo os modelos SD_i e SD_{i+1} chamados de *modelos adjacentes*. Além disso, o modelo SD_{i+1} é um *modelo abstrato* de SD_i.

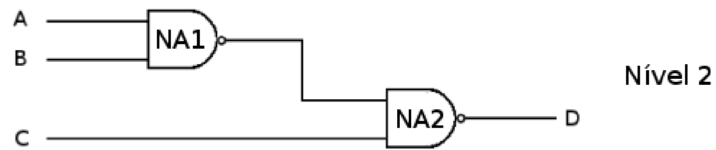
Exemplo 4.1 - Representação em múltiplos níveis de abstração. A Figura 4.1 mostra três modelos representando um circuito digital (circuito que opera com valores 0 e 1). O modelo na Figura 4.1(iii) representa modelo base do circuito, nível 0, e é composto pelos componentes A_1 e A_2, que executam um AND BOOLEANO; e dois inversores, I_1 e I_2, que geram nas suas saídas o valor oposto em relação às suas entradas, isto é, 0 se a entrada for 1, ou vice-versa. O modelo da Figura 4.1(ii) é um modelo abstrato, nível 1, do modelo na Figura 4.1(iii), que utiliza o componente abstrato do tipo NAND, $NA1$, composto pela agregação dos componentes A_1 e I_1. Na Figura 4.1(i) é apresentado outro modelo abstrato para o circuito, nível 2, construído a partir do modelo da Figura 4.1(ii), através da agregação dos componente A_2 e I_2 para gerar o componente abstrato NA_2 do tipo NAND. □

Seja um sistema com k níveis de abstração. Partindo do modelo no nível 0 (isto é, o modelo base), é possível construir uma estrutura hierárquica como mostra a Figura 4.2, sendo que os componentes de um nível i podem ser componentes que já existiam no nível $i-1$ ou componentes abstratos que agregam 1 ou mais componentes do nível $i-1$. Na Figura 4.2, os nós da árvore representam componentes (abstratos ou não); os nós conectados por arestas desenhadas com linhas contínuas correspondem aos componentes de um nível i que foram agregados para definir um componente abstrato no nível $i+1$; os nós conectados por arestas desenhadas com linhas tracejadas, indicam componentes do nível i que também estão presentes no nível $i+1$. As folhas da árvore representam os componentes no modelo base e a raiz representa um componente abstrato que contém o circuito inteiro. Essa representação da raiz da árvore será útil para fazer o diagnóstico, como será mostrado na Seção 4.3.

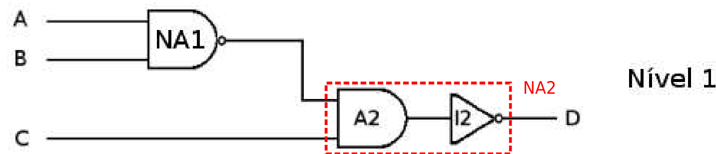
Em um sistema descrito por múltiplos níveis de abstração, todo componente em um nível i deve estar presente no nível $i+1$ ou fazer parte de somente um componente abstrato no nível $i+1$ (suposição de subsistemas independentes). Dessa forma, dados dois componentes abstratos em qualquer nível de abstração, seus conjuntos de componentes agregados devem ser disjuntos (Figura 4.2).

4.1.3 Representação de componentes abstratos

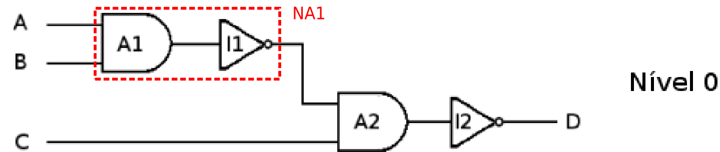
Um componente abstrato deve ser descrito pelos seguintes modelos:



((i)) Modelo abstrato representando o circuito da Figura 4.1(ii), com o componente abstrato NA_2 formando a partir da agregação dos componentes $A2$ e $I2$.



((ii)) Modelo abstrato representando o circuito da Figura 4.1(iii), com o componente abstrato NA_1 formado a partir da agregação dos componente $A1$ e $I1$.



((iii)) Modelo base utilizado para representar um circuito digital.

Figura 4.1: Representação de um circuito digital em diferentes níveis de abstração

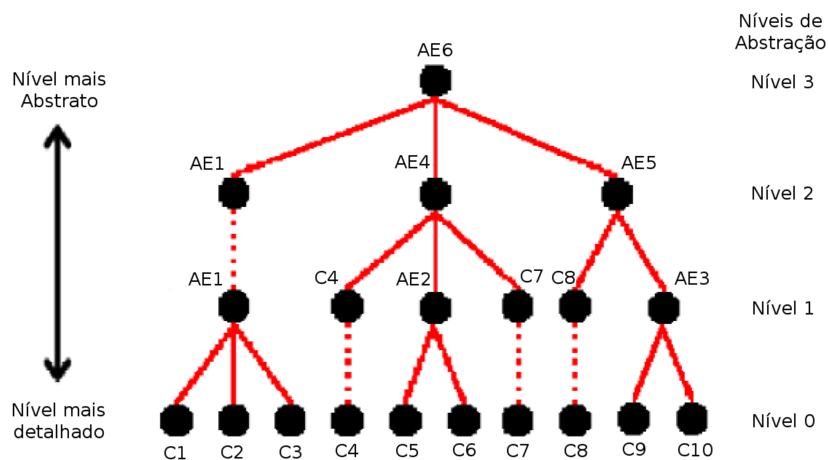


Figura 4.2: Representação da hierarquia de componentes de um sistema descrito em múltiplos níveis de abstração, construída a partir do nível 0. Essa representação é chamada de *Árvore de Abstrações*.

- *modelo estrutural interno*, que descreve como seus componentes agregados estão conectados. Esse modelo é representado apenas pelas sentenças que descrevem as conexões entre dois componentes internos;

- *modelo comportamental*, que descreve como o componente abstrato deve se comportar em termos de seus componentes agregados, isto é, o componente abstrato deve ser modelado de forma que possa representar o comportamento de seus componentes agregados;
- *modelo estrutural externo*, que descreve as conexões do componente abstrato com os outros componentes no mesmo nível de abstração.

Cada um desses modelos usados para descrever um componente abstrato é definido através de um conjunto de sentenças em lógica de primeira ordem.

Exemplo 4.2 - Descrição do modelo base. As seguintes sentenças formam o modelo que descreve o circuito do Exemplo 4.1 (Figura 4.1(iii)):

Modelo comportamental

$$and\!-\!ok(C) \equiv out_1(C) = 1 \leftrightarrow in_1(C) = 1 \wedge in_2(C) = 1$$
$$inv\!-\!ok(C) \equiv out_1(C) = 1 \leftrightarrow in_1(C) = 0$$

$$and(C) \wedge ok(C) \rightarrow and\!-\!ok(C)$$
$$inv(C) \wedge ok(C) \rightarrow inv\!-\!ok(C)$$

$$and(A1)$$
$$and(A2)$$
$$inv(I1)$$
$$inv(I2)$$

Modelo estrutural

$$out_1(A_1) = in_1(I_1)$$
$$out_1(A_2) = in_1(I_2)$$
$$out_1(I_1) = in_1(A_2)$$

□

No Exemplo 4.2, o predicado *and* é usado para representar um tipo de componente que executa um AND BOOLEANO. Assim, os fatos $and(A_1)$ e $and(A_2)$ indicam que os componentes A_1 e A_2 são componentes do tipo AND BOOLEANO. Quando um componente C do tipo *and* está funcionando corretamente, usaremos o predicado $and\!-\!ok(C)$. O predicado *inv* é usado para representar um componente do tipo INVERSOR BOOLEANO. Dessa forma, os fatos $inv(I_1)$ e $inv(I_2)$ indicam que os componentes I_1 e I_2 são do tipo INVERSOR BOOLEANO. De forma análoga ao componente do tipo *and*, quando um componente C do tipo *inv* está funcionando corretamente, usaremos o predicado $inv\!-\!ok(C)$.

O comportamento correto para um componentes do tipo *and* ($and(C)$) é dado em função de suas entradas e saídas e representado por: $and{-}ok(C) \equiv out_1(C) = 1 \leftrightarrow in_1(C) = 1 \wedge in_2(C) = 1$. Nesse caso, a saída 1 do componente C terá o valor 1 somente se as duas entradas forem iguais a 1. Caso contrário a saída será igual a 0. Também, de forma análoga, um componente C do tipo *inv* ($inv(C)$) tem seu comportamento correto dado por $inv{-}ok(C) \equiv out_1(C) = 1 \leftrightarrow in_1(C) = 0$, informando que a saída de um componente C do tipo inversor será igual a 1 somente se a entrada for igual a 0. Essa representação do comportamento correto de um componente será utilizada na descrição de comportamento de componentes abstratos (Exemplo 4.3, a seguir).

O modelo descrito no Exemplo 4.2 também apresenta as conexões entre os componentes (modelo estrutural). Nesse caso, o modelo define que a saída do componente A_1 está ligada à entrada do componente I_1, a saída de I_1 está ligada à entrada 1 de A_2 e a saída de A_2 está ligada à entrada de I_2.

Exemplo 4.3 - Descrição do modelo abstrato. As sentenças a seguir descrevem o modelo abstrato no nível 1 do circuito da Figura 4.1(ii). Note que esse modelo usa parte das sentenças definidas no modelo base do Exemplo 4.2.

Modelo estrutural interno

$$\begin{aligned} & nand{-}composition(NandComp, And, Inv) \rightarrow \\ & \quad in_1(NandComp) = in_1(And) \wedge in_2(NandComp) = in_2(And) \wedge \\ & \quad out_1(And) = in_1(Inv) \wedge out_1(NandComp) = out_1(Inv)) \end{aligned}$$

$$nand{-}composition(NA_1, A_1, I_1)$$

Modelo comportamental

$$\begin{aligned} & nand(C) \wedge ok(C) \rightarrow \exists A, I\, [\, nand{-}composition(C, A, I) \wedge \\ & \qquad and{-}ok(A) \wedge inv{-}ok(I)\,] \end{aligned}$$

$$nand(NA_1)$$

Modelo estrutural externo

$$out_1(NA_1) = in_1(A_2)$$

□

No modelo comportamental do Exemplo 4.3, é definido um componente abstrato do tipo *nand* que agrega dois componente dos tipos *and* e *inv*. O comportamento correto do componente do tipo *nand* é definido em função do comportamento correto de seus componentes agregados. No modelo

estrutural interno, o predicado *nand−composition* é usado para definir como é a composição de um componente do tipo *nand* em função de um componente *and* e um componente do tipo *inv*. Esse modelo, também define os componentes abstratos NA_1, composto dos componentes A_1 e I_1, e NA_2, composto dos componentes A_2 e I_2. O modelo estrutural externo define que a saída do componente NA_1 está conectada à entrada 1 do componente NA_2.

4.2 Solução para um problema de HMBD

Assim como no MBD tradicional, a solução para um problema de HMBD com os modelos abstratos é um conjunto de hipóteses obtidas utilizando um modelo sem componentes (modelo base). Porém, a forma como se chega a essas hipóteses é diferente do MBD. No HMBD, as hipóteses de falha podem ser geradas por chamadas sucessivas a um algoritmo de MBD tradicional (Seção 3.3), considerando as modificações que serão descritas a seguir. A ideia é começar pelo nível mais abstrato e encontrar um diagnóstico para esse nível, sendo essa informação usada para guiar a geração de hipóteses em níveis inferiores [31, 8].

Inicialmente, é feita a *detecção de sintomas*, que funciona de forma semelhante à abordagem tradicional de diagnóstico baseado em modelos usando o modelo base.

A primeira dificuldade que surge no HMBD é sobre os sintomas relacionados às observações em conexões internas do sistema (conexões que não são entradas e nem saídas do sistema). Quando um conjunto de componentes em um nível de abstração i é usado para compor uma abstração em um nível de abstração $i+1$, as observações sobre conexões entre os componentes agregados se escondem por trás daquela abstração, isto é, essas observações não serão consideradas no diagnóstico do nível $i+1$. Essa característica implica numa propriedade importante que surge no diagnóstico hierárquico chamada diagnosticabilidade, que será discutida no final desse capítulo.

Chamaremos de *Diagnóstico por nível de abstração* a técnica de se encontrar as hipóteses de falha em cada nível de abstração do sistema, mas considerando, em cada nível, o resultado obtido em um nível mais abstrato. Para tanto, a sub-tarefa de *geração de hipóteses* terá como responsabilidade gerar as hipóteses de falha em cada nível de abstração do sistema, levando em consideração o resultado do diagnóstico obtido em um nível mais abstrato [31]. Isso é refletido na sub-tarefa de *busca dos contribuintes*, que considera somente um componente C, do nível atual, i, tal que: (1) C pertence a um conjunto de componentes agregados em um componente abstrato CA no nível $i+1$, sendo que CA apareceu em alguma hipótese de falha; ou (2) C é um componente que apareceu em alguma hipótese de falha no nível $i+1$;

4.3 Abordagem *top-down* para fazer o diagnóstico hierárquico

Nessa seção, apresentamos uma abordagem para fazer o diagnóstico hierárquico baseado em modelos, de acordo com a proposta apresentada em [31] e refinada em [8].

Definição 4.1. *Um problema de diagnóstico hierárquico é dado pela tupla* $P_{HMBD} = \langle k, M, COMPS, OBS \rangle$*, sendo que:*

- k *é o número de níveis de abstração utilizados para descrever o sistema;*
- M *é uma sequência de modelos,* $(SD_0, SD_1, \ldots, SD_{k-1})$*, que descrevem o sistema em cada um dos* k *níveis de abstração;*
- $COMPS$ *é uma sequência de conjuntos,* $(COMP_0, COMP_1, \ldots, COMP_{k-1})$*, sendo cada* $COMP_i$ *é um conjunto de constantes representando os componentes presentes no modelo em* SD_i*;*
- OBS *é um conjunto de observações feitas no sistema físico, que serão aplicadas no modelo descrito em* SD_0*.*

Segundo [31,8], a solução para um problema de HMBD pode ser obtida mapeando o problema original em vários subproblemas de MBD tradicional. A Definição 4.2 descreve como esse mapeamento é feito.

Definição 4.2. *Um problema de diagnóstico hierárquico* $P_{HMBD} = \langle k, M, COMPS, OBS \rangle$*, pode ser representado por* k *problemas de MBD tradicional,* $P^i_{MBD} = \langle SD_i, COMP_i, OBS_i \rangle$*, com* $0 \leq i < k$*,* $SD_i \in M$*,* $COMP_i \in COMPS$ *e* $OBS_i \subseteq OBS$ *sendo o conjunto de observações sobre conexões que estão disponíveis no nível* i*.*

Uma abordagem para fazer o diagnóstico hierárquico, chamada de *top-down* [31,8], consiste em encontrar as hipóteses de falha no modelo mais abstrato, nível $k-1$, e utilizar essa informação para guiar a geração de hipóteses do nível inferior, $k-2$, e assim sucessivamente, até se chegar no nível 0. O conjunto de hipóteses de falha encontrado no nível 0 é a solução para o problema de diagnóstico hierárquico. Para tanto, é necessário que seja definida uma relação de comportamento entre os componentes (abstratos ou não) em um nível de abstração i e os componentes no nível abstração $i-1$.

Definição 4.3. *Seja* $P_{HMBD} = \langle k, M, COMPS, OBS \rangle$ *um problema de diagnóstico e* C *um componente pertencente a um nível* $i > 0$*, que tem seu comportamento definido como correto. As seguintes relações de comportamento são válidas:*

- *Se* C *é um componente não abstrato, então o comportamento de* C*, no nível* $i-1$*, deve ser assumido como correto.*
- *Se* C *é um componente abstrato que também pertence ao nível* $i-1$*, então deve ser assumido o comportamento correto de* C *também no nível* $i-1$*.*

- *Se C é um componente abstrato definido no nível i, então todos os componentes agregados de C, referidos por $AGR(C)$, no nível $i-1$, devem ter seus comportamentos assumidos como corretos, conforme o axioma:*

$$ok(CA) \rightarrow \bigwedge_{C \in AGR(CA)} ok(C) \tag{4.1}$$

Dado um problema de diagnóstico hierárquico $P_{HMBD} = \langle k, M, COMPS, OBS \rangle$, e considerando a Definição 4.3, é possível considerar que o espaço de busca usado para encontrar as hipóteses de falha, em cada nível i, seja sempre menor do que o espaço de busca considerando todos os componentes para cada nível (cardinalidade de $COMP_i$).

Uma limitação de se fazer essa relação comportamento (Definição 4.3) surge devido ao Axioma 4.1, que impede a detecção de falhas em componentes abstratos nos casos em que dois componentes falhos usados para compor o componente abstrato anulem a influência um do outro.

Exemplo 4.4 - Relação de comportamento de um componente agregado. Dado o modelo do circuito representado na Figura 4.1(ii), e sabendo que o componente NA_1 está funcionando corretamente ($ok(NA_1)$) então, pelo Axioma 4.1, é possível inferir que os componentes: A_1 e I_1, também têm seus comportamentos assumidos como corretos. □

Na abordagem *top-down*, para cada nível i, é aplicada a relação de comportamento a todos os componentes que apresentam comportamento correto. A ordem na qual cada um dos componentes é selecionado depende somente de uma estratégia particular de implementação. Um dado componente C, em um nível i, tem seu comportamento definido como correto somente se ele não aparece em nenhuma hipótese de falha nesse nível. Assim, dadas as hipóteses de falha de um nível i, antes de fazer a geração de hipóteses no nível $i-1$ é necessário executar os seguintes procedimentos:

1. Para cada componente C, no nível i, que tem seu comportamento definido como correto, aplica-se a relação de comportamento. Durante a geração de hipóteses no nível $i-1$ deve ser considerado que certos componentes já têm seus comportamentos assumidos como corretos e não podem aparecer em nenhuma hipótese de falha.

2. Se um componente C, pertencente ao nível i, faz parte de alguma hipótese de falha nesse nível, então não pode ser aplicada a relação de comportamento e devem ser consideradas as seguintes situações: (1) se C é um componente não abstrato ou C é um componente abstrato que também pertence ao nível $i-1$, então C pode aparecer em hipóteses de falha no nível $i-1$; ou (2) se C é um componente abstrato pertencente ao nível i, cujos componentes agregados pertencem ao nível $i-1$, então esses componentes agregados podem aparecer em hipóteses no nível $i-1$.

Usando os procedimento citados, a geração de hipóteses no nível de abstração 0 será feita considerando que parte dos componentes em $COMP_0$ terá seu comportamento assumido como correto, devido às informações obtidas nos níveis mais abstratos. Assim, a tarefa de busca de contribuintes para esse nível terá que lidar com um número menor de componentes do que teria caso fosse utilizada abordagem de MBD tradicional diretamente no modelo base.

4.4 Algoritmo para fazer o diagnóstico hierárquico

Nessa seção, são apresentados os algoritmos para fazer diagnóstico hierárquico, de acordo com os conceitos apresentados na Seção 4.3.

O Algoritmo 5 é utilizado para fazer o diagnóstico hierárquico. A entrada para esse algoritmo é um problema de diagnóstico hierárquico $\langle k, M, COMPS, OBS \rangle$, sendo OBS um conjunto de observações feitas no sistema físico, ou seja, no nível 0. Um processamento inicial, Algoritmo 3, verifica quais conexões que possuem valores observados estão disponíveis para serem usadas em cada um dos níveis de abstração (conexões que não foram escondidas em componentes abstratos). A função *Abstract* (Linha 4) é responsável por verificar quais observações de um nível i podem ser usadas em um nível $i+1$. Para cada nível i é definido uma variável, OBS_i, que conterá as observações de cada nível. Ao final, o Algoritmo 3 devolve o nível mais alto para o qual foi possível mapear as observações e as observações de cada nível. Todos os níveis para os quais não foi possível mapear observações devem ser desconsiderados durante o processo, visto que não será possível reconhecer nenhuma hipótese de falha nesses níveis.

O Algoritmo 4 é responsável por encontrar a solução para o problema de diagnóstico hierárquico através de uma sequência de chamadas ao algoritmo de diagnóstico tradicional (função MBD (Linha 3). O algoritmo implementado na função MBD deve verificar quais componentes tiveram seus comportamentos assumidos como corretos previamente. Nesse ponto, Δ contém o diagnóstico para o sistema no nível i, que será representado como: Δ_i. Caso ainda exista um nível menor que i, serão aplicadas as relações de comportamento (Definição 4.3) em todos os componentes envolvidos em alguma hipótese de falha. Esse processo é repetido sucessivamente até chegar no nível 0. As hipóteses de falha encontradas nesse nível são devolvidas como solução para o problema de diagnóstico hierárquico.

4.5 Exemplo de HMBD

Para demonstrar o funcionamento do Algoritmo 5, utilizaremos o circuito definido nas Figuras 4.1(iii) e 4.1(ii). Seja o problema de diagnóstico hierárquico $P_{HMBD} = \langle k, M, COMPS, OBS \rangle$, com $k = 2$; $M = \{SD_0, SD_1\}$, sendo SD_0 o modelo base (descrito no Exemplo 4.2), SD_1 um modelo abstrato de SD_0 (descrito no Exemplo 4.3); $COMPS = \{\{A_1, A_2, I_1, I_2\}, \{NA_1, A_2, I_2\}\}$; e $OBS = \{A = 0, B = 1, C = 0, D = 0\}$. Pelas observações é possível verificar que o sistema apresenta falha.

Algoritmo 3: `Bottom-Up-Abstract-Observations`($\langle k, M, COMPS, OBS \rangle$)

Entrada: $\langle k, M, COMPS, OBS \rangle$: um problema de diagnóstico hierárquico, sendo: $M = (SD_0, \ldots, SD_{k-1})$ e $COMPS = (COMP_0, \ldots, COMP_{k-1})$.

Saída: o nível mais alto que contém observações e as observações acessíveis em cada nível i, disponibilizadas em OBS_i.

1 $OBS_0 = OBS$;
2 $l \leftarrow 0$;
3 **enquanto** $l < k - 1$ *e* $OBS_l \neq \emptyset$ **faça**
4 $\quad OBS_{l+1} \leftarrow Abstract(OBS_l, SD_{l+1}, COMP_{l+1})$;
5 $\quad l \leftarrow l + 1$;
6 **fim**
7 **se** $OBS_l = \emptyset$ **então**
8 $\quad l \leftarrow l - 1$;
9 **devolva** l *e* $(OBS_0, \ldots, OBS_{k-1})$

Algoritmo 4: `Top-Down-Diagnosis`($\langle k, M, COMPS, OBS \rangle$, l, $OBS^\star$)

Entrada: $\langle k, M, COMPS, OBS \rangle$: um problema de diagnóstico hierárquico, sendo: $M = (SD_0, \ldots, SD_{k-1})$ e $COMPS = (COMP_0, \ldots, COMP_{k-1})$; l: o nível mais alto contendo conexões com observações; $OBS^\star$: conjunto de observações para conexões em cada nível i, $0 \leq i \leq l$.

Saída: Hipóteses de falha para o problema $P_{MBD} = \langle SD_0, COMP_0, OBS_0 \rangle$.

1 $i \leftarrow l$;
2 **enquanto** $i \geq 0$ **faça**
3 $\quad \Delta \leftarrow MBD\,(\langle SD_i, COMP_i, OBS_i \rangle)$;
4 $\quad$ **se** $i > 0$ **então**
5 $\quad\quad CompsFalhos \leftarrow \bigcup_{\Delta' \in \Delta} \Delta'$;
6 $\quad\quad$ **para cada** $C \in (COMP_i - CompsFalhos)$ **faça**
7 $\quad\quad\quad$ aplicar as relações de comportamento (Definição 4.3) ;
8 $\quad\quad$ **fim**
9 $\quad i \leftarrow i - 1$;
10 **fim**
11 **devolva** Δ ;

No início, o Algoritmo 3 é executado para disponibilizar as observações em cada nível de abstração. Nesse caso, $OBS_0 = OBS$, e como todas as conexões com valores observados estão disponíveis no modelo do nível 1, então, obtém-se que: $OBS_1 = OBS_0$. O Algoritmo 5 inicia no nível 1, e encontra as hipóteses de falha nesse nível (Linha 3), devolvendo $\Delta_1 = \{A_2, I_2\}$. Como NA_1 não aparece em nenhuma hipótese de falha, é aplicada a relação de comportamento (Linha 7) fazendo com que os componentes A_1 e I_1, no nível 0, tenham seus comportamentos assumidos como correto. Fazendo o diagnóstico no nível 0, obtém-se $\Delta_0 = \{A_2, I_2\}$, como sendo a solução para o problema de diagnóstico hierárquico.

4.6 Desempenho do HMBD

A vantagem de se utilizar o HMBD vem do fato de ser possível utilizar um modelo mais simples, com menos componentes, para encontrar as hipóteses de falha nos níveis menos abstratos, assu-

Algoritmo 5: `HierarchicalDiagnosis`$(\langle k, M, COMPS, OBS \rangle)$

Entrada: $\langle k, M, COMPS, OBS \rangle$: um problema de diagnóstico hierárquico.
Saída: Hipóteses de falha para o problema $P_{MBD} = \langle SD_0, COMP_0, OBS_0 \rangle$.

1 $l, OBS^\star \leftarrow$ Abstract-Observations$(\langle k, M, COMPS, OBS \rangle))$;
2 $\Delta \leftarrow$ Top-Down-Diagnosis$(\langle k, M, COMPS, OBS \rangle$, l, $OBS^\star)$;
3 **devolva** Δ ;

mindo que é possível desconsiderar uma parte desses componentes durante a geração de hipóteses (componentes que têm seus comportamentos assumidos como corretos em níveis mais abstratos). Apesar dessa vantagem, em certas situações, é possível levar mais tempo para encontrar a solução para um problema de diagnóstico hierárquico do que usando a abordagem de diagnóstico baseado em modelos tradicional. Por exemplo, se todos os componentes do nível mais abstrato de um sistema pertencem a alguma hipótese de falha, não será possível inferir que algum componente no nível mais adjacente menos abstrato está funcionando corretamente. Caso isso ocorra sucessivamente, até chegar no nível 0, a técnica de diagnóstico hierárquico apresentará um desempenho muito pior que a técnica de diagnóstico baseado em modelos tradicional.

Por outro lado, as abstrações permitem que grandes partes do sistema seja isoladas logo de início. Por exemplo, se um sistema é definido em dois níveis de abstração, sendo o modelo base composto de cinco componentes, $\{C_1, C_2, C_3, C_4, C_5\}$, e o modelo no nível abstrato composto de dois componentes, sendo CA_1 um componente composto da agregação: $\{C_1, C_2, C_3\}$, e CA_2 outro componente composto da agregação: $\{C_4, C_5\}$. Suponha que ao fazer o diagnóstico no nível 1, obtenha-se $\Delta_1 = \{CA_2\}$, e em seguida, fazendo o diagnóstico no nível 0 obtenha-se $\Delta_0 = \{C_4, C_5\}$. Nesse caso, o espaço de busca envolvido para encontrar as hipóteses de falha é de tamanho: $2^2 + 2^3 = 12$ (2^2 para o nível 1 e 2^3 para o nível 0, considerando que os componentes C_1, C_2 e C_3 apresentam um comportamento correto). Por outro lado, fazendo o diagnóstico direto no modelo base, o espaço de busca considerado é de tamanho: $2^5 = 32$.

4.7 Considerações finais

Apesar do algoritmo de diagnóstico hierárquico proposto em [31, 8] (Algoritmo 5) encontrar a solução para um problema de diagnóstico hierárquico, a forma como ele é definido traz alguns efeitos indesejáveis, tais como:

- O modelo usado para fazer o diagnóstico sempre coincide com o modelo definido em um dos níveis da árvore de abstrações. Essa abordagem impede que sejam adotadas estratégias que definem quais componentes abstratos devem ser substituídos pelos seus componentes agregados (e suas conexões) e a ordem que essas substituições são feitas. Por exemplo, se um componente abstrato CA, que aparece em um nível i, não está envolvido em nenhuma hipótese de falha nesse nível, então não é necessário substituí-lo pelos seus agregados nos níveis menos abstratos, fazendo com o espaço de busca envolvido na geração de hipóteses dos

níveis menos abstratos seja reduzido.

- O algoritmo somente finaliza após fazer o diagnóstico usando o modelo base, nível 0. Dessa forma, mesmo que seja possível obter a solução para o problema de diagnóstico hierárquico em um modelo mais abstrato, o algoritmo continuará a sua execução sem necessidade. Um exemplo desse comportamento pode ser visto quando é dado um sistema com $k > 1$ níveis de abstrações, e o diagnóstico obtido em um nível i, $0 < i < k$, contém somente componentes que não são abstratos. Assim, não há necessidade de se fazer o diagnóstico nos níveis menos abstratos.

- Suponha que CA seja um componente abstrato que foi definido em um nível $i - 2$ e que, durante a execução do Algoritmo 4, apareça em pelos menos uma hipótese de falha no nível $i - 1$. Nesse algoritmo, quando for feito o diagnóstico no nível $i - 2$ ainda será utilizado o componente abstrato CA, mas ao invés disso, esse componente já poderia ser substituído pelos seus componentes agregados e suas conexões. Com essa modificação, seria possível chegar antes aos componentes não abstratos que estão falhos, sem ter que sempre chegar a um modelo que não contenha componentes abstratos para dar a solução.

- A solução para o problema de diagnóstico hierárquico, como apresentado, pode levar à não diagnosticabilidade do sistema. Isso ocorre porque não há nenhum critério para definir quais as conexões que podem ter seus valores observados. Dessa forma se não for possível detectar nenhum sintoma no nível de abstração devolvido pelo Algoritmo 3 (por exemplo, nos casos quando existe somente uma observação disponível), não será possível fazer o diagnóstico.

No Capítulo 6, apresentaremos uma extensão dessa abordagem *top-down* (que também será utilizada para fazer depuração de programas considerando abstrações), que permite tratar essas questões apontadas nessa seção.

No próximo capítulo, apresentaremos uma abordagem para fazer a depuração de programas utilizando a técnica de diagnóstico baseado em modelos.

Capítulo 5

Depuração de Programas Baseada em Modelos

Da mesma forma que um engenheiro procura pelos componentes falhos de um sistema físico, confrontando o projeto desse sistema (modelo) com o comportamento observado no sistema real, um programador confronta as suas intenções com o seu programa para encontrar as sentenças falhas. Nesse capítulo, mostraremos que a verificação das diferenças entre as intenções do programador o comportamento do seu programa definem o problema de depuração de programas como um problema de diagnóstico baseado em modelos.

5.1 Diagnóstico *versus* depuração

Diagnóstico baseado em modelos para depuração de programas, também chamado de *depuração de programas baseada em modelos* (*Model based software debugging* - MBSD), foi originalmente proposto por Console [9], com o objetivo de encontrar falhas em programas lógicos. No projeto JADE [29], essa técnica é usada para auxiliar programadores experientes a encontrarem falhas em programas escritos na linguagem Java [26, 27, 46]. O sistema ProPAT [15, 3] se baseou no projeto JADE, porém é voltado para o ensino/aprendizagem de programação na linguagem C.

No diagnóstico baseado em modelos tradicional para sistemas físicos, o modelo descreve o comportamento correto do sistema e as observações feitas no sistema físico refletem o seu comportamento observado, possivelmente falho. Na depuração de programas baseada em modelos, o modelo é derivado a partir do programa (*modelo do programa*), com possíveis falhas, e as observações são obtidas a partir de entradas e saídas esperadas, por exemplo, definidas por *casos de teste*. As hipóteses de falha no MBSD são compostas pelas linhas do programa que podem estar falhas (por exemplo, uma variável que foi iniciada de maneira incorreta; uma expressão errada ou ainda uma condição incorreta de uma estrutura de repetição).

É importante notar que essa técnica de depuração não requer que seja fornecida uma especificação formal do comportamento esperado do programa, sendo necessário somente: (1) o próprio programa; (2) os casos de teste e (3) uma descrição do comportamento correto dos comandos da linguagem. Com base nas descrições em (3), a construção do modelo a partir do programa é feita

de forma automática.

Um aspecto importante é que enquanto as predições no MBD para sistemas físicos são feitas por um sistema de diagnóstico, no MBSD as predições são feitas pelo próprio programador, com base em seu modelo mental. Por exemplo, um programador consegue predizer que o fatorial do número 4 é 24, o fatorial de 5 é 120, e assim por diante. Dessa forma, o programador deve ser capaz de comunicar suas intenções utilizando as instruções da linguagem de programação, de maneira que sejam satisfeitas as metas e as submetas para o problema que está sendo resolvido.

As principais diferenças entre o diagnóstico baseado em modelos para sistemas físicos e a depuração de programas baseada em modelos são sintetizadas na Tabela 5.1 [15].

	Diagnóstico baseado em modelos de sistemas físicos	**Diagnóstico baseado em modelos de programas**
Modelo do Sistema	modelo correto do sistema	modelo do programa com possíveis erros de programação
Observações	medidas no sistema físico que refletem o comportamento (incorreto) do sistema com falhas	entradas e saídas calculadas pelo programa com falhas e que refletem o seu comportamento (incorreto)
Predições	predições feitas pelo sistema de diagnóstico sobre o modelo correto do sistema	predições feitas pelo programador sobre o seu modelo (mental) da solução correta
Hipóteses de falha	composta dos componentes falhos do sistema	composta das linhas falhas do programa

Tabela 5.1: Diferenças entre a abordagem de diagnóstico baseado em modelos para sistemas físicos e a depuração de programas baseada em modelos [15].

Para fazer a depuração de programas baseada em modelos podem ser usados dois tipos de modelos:

- *modelo baseado em dependência* [26, 46], que constrói um modelo baseado em um grafo de dependências entre as variáveis do programa;
- *modelo baseado em valor* [27], que constrói o modelo baseado nas variáveis que as sentenças e expressões do programa usam e modificam.

Nesse trabalho, utilizaremos somente o modelo baseado em valor pois esse modelo permite detectar vários tipos de falhas que não detectadas pelo modelo baseado em dependência.

A seção a seguir faz uma introdução à terminologia de depuração de programas que será utilizada no decorrer desse capítulo e apresenta uma taxionomia de falhas, usada para definir os tipos de falhas para as quais o MBSD pode ser usado.

5.2 Depuração de programas baseada em modelos

As tarefas realizadas no MBSD são as mesmas do processo de diagnóstico baseado em modelos de sistemas físicos (Seção 3.3), sendo a principal diferença a construção do modelo. Para cada

programa, possivelmente, com defeito, um novo modelo é construído de forma automática (descrito na Seção 5.4). As sub-tarefas do processo de diagnóstico descritas no contexto de depuração de programas são:

1. *Detecção de sintomas*. Consiste em executar os casos de teste definidos para o problema de programação em questão, verificando quais produzem erro. Um sintoma é caracterizado por um valor de saída do programa que é diferente do valor esperado, conforme algum caso de teste. Se vários casos de teste produzirem erro, um deles é selecionado para fornecer as entradas e saídas usadas no processo de depuração.

2. *Geração de hipóteses*, consiste em gerar um conjunto de hipóteses sobre as linhas falhas do programa. Essa sub-tarefa é dividida em três sub-tarefas:

   (a) *Busca de contribuintes*, em que são encontrados os conjuntos de componentes contribuintes para as falhas. Os componentes de um conjunto de contribuintes representam as sentenças e expressões do programa que não podem ser consideradas corretas ao mesmo tempo. Dado o modelo construído a partir do programa, é possível encontrar os conjuntos de contribuintes utilizando as mesmas abordagens citadas na Seção 3.6, isto é, um mecanismo de propagação de restrições ou um provador de teoremas;

   (b) *Transformação de conjuntos de contribuintes em hipóteses*. Dado um conjunto de linhas que contribuem para ocorrência de falhas, o Algoritmo de Reiter (Algoritmo 3.6) é usado para transformá-lo em conjuntos de hipóteses de falha. Cada componente em uma hipótese de falha corresponde a uma linha do programa que contém a sentença ou a expressão que pode estar com falha;

   (c) *Predição baseada em filtragem*, que usa somente a informação disponível no modelo do programa para reduzir o conjunto das hipóteses.

3. *Discriminação de hipóteses*, em que novas observações a respeito do comportamento esperado do programa são fornecidas pelo programador, com o objetivo de reduzir o número de hipóteses. As novas observações consistem de valores esperados para as variáveis em diversos pontos do programa.

5.3 Modelo Baseado em Valor

No modelo baseado em valor (*Value Based Model* - VBM) [27], expressões e sentenças do programa são representadas por componentes, enquanto variáveis são representadas como conexões entre componentes. Dois componentes estão conectados somente se existe um fluxo de informação entre as respectivas sentenças ou expressões que eles representam. Um fluxo de informação entre sentenças do programa é definido quando uma variável v tem seu valor atribuído em uma sentença

S_1 e posteriormente utilizada em outra sentença ou expressão S_2, não existindo entre S_1 e S_2 nenhuma sentença que modifique o valor de v.

Os modelos estrutural e comportamental dos componentes no VBM são descritos com sentenças da lógica de primeira ordem. Seja C um componente do modelo que representa uma sentença S do programa. Para cada variável utilizada em S é criada uma porta de entrada em C e para cada variável que S modifica é criada uma porta de saída em C. O conjunto de portas de entrada do componente C é representado por $input(C)$ e o conjunto de portas de saída de C é representado por $output(C)$. A $i - \acute{e}sima$ porta de entrada do componente C e a $j - \acute{e}sima$ porta de saída do componente C são representadas, respectivamente, por $in_i(C)$ e $out_j(C)$. Quando um componente C possui somente uma porta de entrada (respectivamente para a saída), esta será representada por $in(C)$ (respectivamente $out(C)$).

Alguns componentes podem ter portas com um significado especial, por exemplo: o componente que representa o comando de seleção simples (`if`) tem uma porta de entrada, *cond*, que representa o resultado da avaliação da expressão utilizada na condição. Outros componentes são compostos por subsistemas. Por exemplo, um componente de seleção alternativa (*if-then-else*) é composto de um subsistema representando as sentenças do ramo *then* e outro subsistema representando as sentenças no ramo *else*.

As seções a seguir apresentam como as sentenças e as expressões de uma linguagem de programação são mapeadas em componentes e conexões de um modelo baseado em valor, conforme descrito em [27]. Esses mapeamentos abrangem somente programas escritos como um subconjunto da linguagem Java, gerado a partir da gramática, descrita em BNF, na Figura 5.1.

5.3.1 Expressões

Expressões são construções usadas em sentenças, mas que não podem ser sentenças por si próprias (exceto chamadas a métodos que devolvem valores). As expressões são divididas nos seguintes tipos: *constantes*, *variáveis*, *chamadas de métodos que devolvem valores* e *expressões com operadores binários*. As subseções a seguir descrevem cada um dos tipos de expressões, com exceção das chamadas de métodos que devolvem valores que serão apresentadas na Seção 5.3.7.

Expressões dos tipos: Constantes e Variáveis

Sintaxe para constantes:	<Expr> ::= <Const>
Sintaxe para variáveis :	<Expr> ::= <Id>

Constantes e variáveis são mapeadas em conexões. Uma conexão representando uma constante assume um valor fixo, relacionado ao valor da constante no programa. Uma conexão representando uma variável é construída na inicialização da variável e disponibilizada para uso posterior, sendo

⟨Classes⟩ ::= ⟨Class⟩ ⟨Classes⟩ | ϵ
⟨Class⟩ ::= `class` ⟨IdClass⟩ {⟨ClassStmts⟩}
⟨ClassStmts⟩ ::= ⟨ClassStmt⟩ ⟨ClassStmts⟩ | ϵ
⟨ClassStmt⟩ ::= ⟨VariableDecl⟩`;` | ⟨MethodDecl⟩
⟨VariableDecl⟩ ::= ⟨Type⟩ ⟨Id⟩
⟨Type⟩ ::= `int` | `boolean` | ⟨IdClass⟩
⟨TypeV⟩ ::= `void` | ⟨Type⟩
⟨MethodDecl⟩ ::= ⟨TypeV⟩ ⟨Id⟩ `(`⟨ParList⟩`)` {⟨JavaStmts⟩}
⟨ParList⟩ ::= ⟨VariableDecl⟩ ⟨ParListRest⟩ | ϵ
⟨ParListRest⟩ ::= `,` ⟨VariableDecl⟩ ⟨ParListRest⟩ | ϵ
⟨JavaStmts⟩ ::= ⟨JavaStmt⟩ ⟨JavaStmts⟩ | ϵ
⟨JavaStmt⟩ ::= ⟨Assignment⟩ | ⟨Selection⟩ | ⟨While⟩ | ⟨MethodCallV⟩ |
  ⟨ReturnStmt⟩
⟨Assignment⟩ ::= ⟨Id⟩ `=` ⟨Expr⟩
⟨Expr⟩ ::= ⟨Id⟩ | ⟨Const⟩ | ⟨MethodCall⟩ | `(`⟨Expr⟩`)` |
  ⟨Expr$_1$⟩ ⟨Op⟩ ⟨Expr$_2$⟩
⟨Selection⟩ ::= `if` `(`⟨Expr⟩`)` {⟨ThenStmts⟩} ⟨ElseStmts⟩
⟨ThenStmts⟩ ::= ⟨JavaStmts⟩
⟨ElseStmts⟩ ::= `else` {⟨JavaStmts⟩} | ϵ
⟨While⟩ ::= `while` `(`⟨Expr⟩`)` {⟨JavaStmts⟩}
⟨MethodCall⟩ ::= ⟨Id⟩ `(`⟨ActualParList⟩`)` | ⟨IdClass⟩`.`⟨Id⟩`(`⟨ActualParList⟩`)`
⟨MethodCallV⟩ ::= ⟨Id⟩ `(`⟨ActualParList⟩`)` | ⟨IdClass⟩`.`⟨Id⟩`(`⟨ActualParList⟩`)`
⟨ActualParList⟩ ::= ⟨Expr⟩ ⟨ActualParListRest⟩ | ϵ
⟨ActualParListRest⟩ ::= `,` ⟨Expr⟩ ⟨ActualParListRest⟩ | ϵ
⟨ReturnStmt⟩ ::= `return` ⟨Expr⟩`;`

Figura 5.1: Gramática BNF definindo um subconjunto da linguagem Java. Os elementos <Id>, <IdClass> e <Const> representam identificadores para a linguagem e <Op> representa um dos operadores binários: $+, -, *, /, \%, <, >, <=, >=, ==, !=, \&\&, ||$.

que novas conexões para uma mesma variável podem ser criadas conforme o valor dessa variável é modificado (mais detalhes na Seção 5.3.2).

Expressões do tipo operador binário

Sintaxe: <Expr> ::= <Expr$_1$ > <Op> <Expr$_2$ >

Expressões do tipo operador binário são mapeadas em componentes do tipo *Expr* com duas portas de entrada e uma porta de saída. Seja C um componente do tipo *Expr*, representando a expressão definida por <Expr>. A porta de saída $result(C)$ representa o valor da avaliação da expressão. As portas de entrada, $in_1(C)$ e $in_2(C)$, são conectadas às portas $result$ dos componentes representando as expressões <Expr$_1$> e <Expr$_2$>.

Seja $Op(in_1, in_2, out)$ um predicado que representa uma operação binária válida na linguagem, in_1 e in_2 os operandos de Op e out representa o resultado da expressão ($out = in_1 \;\; Op \;\; in_2$). O

modelo comportamental de C é definido como:

$$ok(C) \rightarrow Op(in_1(C), in_2(C), result(C))$$

sendo Op substituído pelo predicado responsável pela operação em questão.

5.3.2 Atribuições

Sintaxe: <Assignment> ::= <Id> = <Expr>

Atribuições são mapeadas em componentes do tipo *Assignment* que possuem uma porta de entrada e uma porta de saída. Seja C um componente do tipo *Assignment*. A porta de entrada $in(C)$ deve ser conectada à porta $result$ do componente que representa a expressão no lado direito da sentença de atribuição (<Expr>) e conectada à porta de saída $out(C)$. A porta de saída está relacionada com a variável no qual o valor será atribuído. Sempre que uma variável v é usada no lado esquerdo de uma atribuição, uma nova conexão representando o novo valor de v é criada. Dessa forma, sempre que um componente precisar usar a variável v, haverá uma conexão para v representando o seu valor atual.

O modelo comportamental desse tipo de componente é descrito como:

$$ok(C) \rightarrow out(C) = in(C)$$

5.3.3 Seleções

Sintaxe: <Selection> ::= `if (`<Expr>`)` {<ThenStmts>} <ElseStmts>

Seleções são mapeadas em componentes do tipo *Conditional* e um componente do tipo *Expr* que representa a expressão condicional da seleção (<Expr>). O componente *Conditional* possui um número variável de portas de entrada e portas de saída, além de uma porta de entrada, *cond*, que representa o resultado da avaliação da expressão condicional. São definidos dois subsistemas no componente *Conditional*, sendo um para o ramo *then* (C_then) e outro para o ramo *else* (C_else). Para cada variável usada em qualquer um dos ramos é criada uma porta de entrada no componente de seleção e para cada variável modificada em qualquer um dos ramos é criada uma porta de saída. Se C é um componente do tipo *Conditional*, $input(C)$ representa o conjunto de portas de entrada de C e $output(C)$ representa o conjunto das portas de saída de C.

Cada subsistema contém um conjunto de portas de entrada e saída que coincidem, respectivamente, com as portas de entrada e saída do componente de seleção (exceto pela porta *cond*), mas que não estão conectadas. Dependendo do valor em $cond(C)$, um dos subsistemas é executado, e as portas de entrada e saída de C são conectadas às portas do subsistema.

O comportamento de um componente C do tipo $Conditional$ é definido como:

$$ok(C) \wedge cond(C) = true \rightarrow \forall k \mid in_k(C) \in input(C)\ ,\ in_k(C) = in_k(C_then)$$
$$ok(C) \wedge cond(C) = true \rightarrow \forall k \mid out_k(C) \in output(C)\ ,\ out_k(C) = out_k(C_then)$$
$$ok(C) \wedge cond(C) = false \rightarrow \forall k \mid in_k(C) \in input(C)\ ,\ in_k(C) = in_k(C_else)$$
$$ok(C) \wedge cond(C) = false \rightarrow \forall k \mid out_k(C) \in output(C)\ ,\ out_k(C) = out_k(C_else)$$

5.3.4 Repetições

Sintaxe: <While> ::= `while(`<Expr>`)` {<JavaStmts> }

Repetições são transformadas em seleções aninhadas e mapeadas em componentes do tipo $Conditional$. O número de seleções aninhadas para cada laço depende do número de iterações que o comando de repetição executa, para o caso de teste que forneceu as entradas e saídas usadas para fazer a depuração. O modelo gerado a partir da substituição do comando de repetição por uma sequência de seleções aninhadas é chamado de *Modelo Livre de Laço* (LFM - *Loop Free Model*) [30].

5.3.5 Comando return

Sintaxe: <ReturnStmt> ::= `return` <Expr>;

O comando *return* é mapeado para um componente do tipo *Return*, contendo uma porta de entrada e uma porta de saída. A porta de entrada está relacionada com a porta *result* do componente que representa a expressão avaliada (<Expr>) no comando `return` e a porta de saída, *return*, disponibiliza o valor da expressão.

O modelo comportamental de um componente C do tipo *Return* é definido como:

$$ok(C) \rightarrow return(C) = in(C)$$

5.3.6 Chamada a métodos que não devolvem valores

Sintaxe: <MethodCallV> ::= <Id> (<ActualParList>) |
<IdClass>.<Id> (<ActualParList>)

Métodos que não devolvem valores (chamados de procedimentos nesse texto), podem modificar atributos definidos na mesma classe na qual esses métodos foram declarados ou atributos dos objetos passados como parâmetros. As chamadas para métodos desse tipo são sentenças por si próprias e são mapeadas em componentes do tipo $MethodCallV$, contendo várias portas de entrada e saída. As portas de entrada estão relacionadas com os atributos do objeto usados pelo método e os parâmetros passados na chamada, enquanto as portas de saída estão relacionadas com os atributos do objeto e os parâmetros que foram modificados no corpo do método.

Seja m um método chamado no programa e M o modelo representando esse método. Um componente MC representando a chamada para m possui o mesmo número de portas de entrada e saída utilizadas em M, que estão relacionadas às variáveis e objetos utilizados e modificados em m.

5.3.7 Chamada a métodos que devolvem valores

Sintaxe:	<MethodCall> ::= <Id> (<ActualParList>) |
	<IdClass>.<Id> (<ActualParList>)

Chamadas a métodos que devolvem valores (chamados de funções nesse texto) são mapeados em componentes do tipo $MethodCall$, semelhantes aos componentes do tipo $MethodCallV$, com a adição de uma porta *result* representando o valor devolvido na chamada do método. O modelo descrevendo um método que devolve valor deve ter um componente do tipo *Return* com a sua porta de saída conectada à porta *result* do componente representando a chamada do método.

O modelo comportamental de um componente do tipo $MethodCall$ é construído de forma semelhante à feita no componente $MethodCallV$.

5.4 Transformação do programa em um modelo baseado em valor

O modelo baseado em valor de um programa é construído com base na árvore sintática abstrata (AST - *Abstract Syntax Tree*) do programa, gerada por um processo de análise sintática. Uma vez gerada, a AST é percorrida em pré-ordem, criando os componentes do modelo de acordo com os mapeamentos apresentados na Seção 5.3.

Algoritmo 6: `CreateModel(`(*Program*)

Entrada: **Program**: o programa do aluno.
Saída: o modelo baseado em valor para *Program*.

```
1  Criar a AST para Program ;
2  percorrer a AST em pré-ordem
3      para cada nó n da AST faça
4          criar o componente representando n ;
5          identificar as variáveis usadas e modificadas pelos componentes ;
6      fim
7  fim
8  percorrer a AST em pré-ordem
9      para cada nó n da AST faça
10         criar novas versões para todas as variáveis modificadas ;
11         conectar as novas variáveis aos componentes correspondentes ;
12     fim
13 fim
14 devolva o VBM de Program
```

O Algoritmo 6, apresentado em [15], é utilizado para construir o modelo baseado em valor a

partir do programa do aluno. Esse algoritmo constrói o modelo em duas fases. Na primeira fase (bloco definido na Linha 2) são construídos os componentes do modelo e verificadas quais variáveis cada componente usa e modifica. Na segunda fase (bloco definido na Linha 8) são criadas as novas versões para as variáveis que são modificadas pelos componente e conectadas com os componentes correspondentes. Ao final, o modelo construído com essas operações é devolvido.

5.5 Exemplo de diagnóstico com MBSD

Suponha que um aprendiz de programação tenha construído um programa na linguagem Java para resolver o problema do *módulo de um inteiro*: *Dado um número inteiro n, encontrar o módulo desse número. O módulo de um inteiro n é: n, se $n \geq 0$; ou $-n$, se $n < 0$.* A Figura 5.2 mostra um método extraído do programa do aluno, responsável por solucionar o problema. Analisando o método é possível notar que existem duas possíveis falhas: (i) a condição na Linha 4 deveria ser $valor > 0$ (falha funcional) ou (ii) as Linhas 5 e 7 estão invertidas (falha estrutural).

```
1   public int modulo(int valor) {
2       int m;
3
4       if (valor < 0) {
5           m = valor;
6       } else {
7           m = valor * -1;
8       }
9
10      return m;
11  }
```

Figura 5.2: Método `modulo` escrito por um aprendiz de programação, na linguagem Java, para encontrar o módulo de um número inteiro. Note que o método apresenta uma ou mais falhas.

Mostraremos como fazer a depuração de programas baseada em modelos para encontrar as possíveis falhas no método da Figura 5.2. Note que somente o método será depurado. As entradas e saídas dos casos de teste serão, respectivamente, os parâmetros de entrada para o método e o valor devolvido pelo método.

5.5.1 Detecção de sintomas

Na Tabela 5.2 são apresentados alguns casos de teste para o método `modulo`, que serão utilizados na detecção de sintomas. Para os casos de teste 2 e 3 o método devolve valores diferentes dos esperados. No caso de teste 2 o valor esperado é 5, mas o método devolve o valor -5; enquanto que, no caso de teste 3 o valor esperado é 3, e o método devolve -3. Essas discrepâncias entre os valores esperados e os valores devolvidos pelo método caracterizam os sintomas.

Como dois casos de teste falharam, um desses deve ser escolhido para continuar o processo

Caso de Teste	Entrada	Saída
1	0	0
2	5	5
3	-3	3

Tabela 5.2: Casos de testes usados para encontrar falhas no método da Figura 5.2. A coluna *Caso de Teste* é o identificador do caso de teste, as colunas *Entrada* e *Saída* definem, respectivamente, o parâmetro de entrada e o valor de saída esperado para o método.

de depuração. Para esse exemplo, o caso de teste escolhido é o de número 2. Na seção seguinte apresentamos o modelo que representa esse método.

5.5.2 Construção do modelo baseado em valor

Como foram detectados sintomas no método `modulo`, será construído um modelo para representar esse método, utilizando a abordagem apresentada na Seção 5.4.

A Figura 5.3 mostra uma representação para o modelo estrutural do método `modulo`. O modelo é composto dos componentes $\{C_1, C_2, \ldots, C_8\}$, cada um representando uma sentença ou expressão usada em uma das linhas do método. Por exemplo: o componente C_2 representa o comando de seleção alternativa *if-then-else* (linhas 4 a 8) e os componentes C_3 e C_4 representam os subsistemas de C_2, os ramos *then* e *else*, respectivamente.

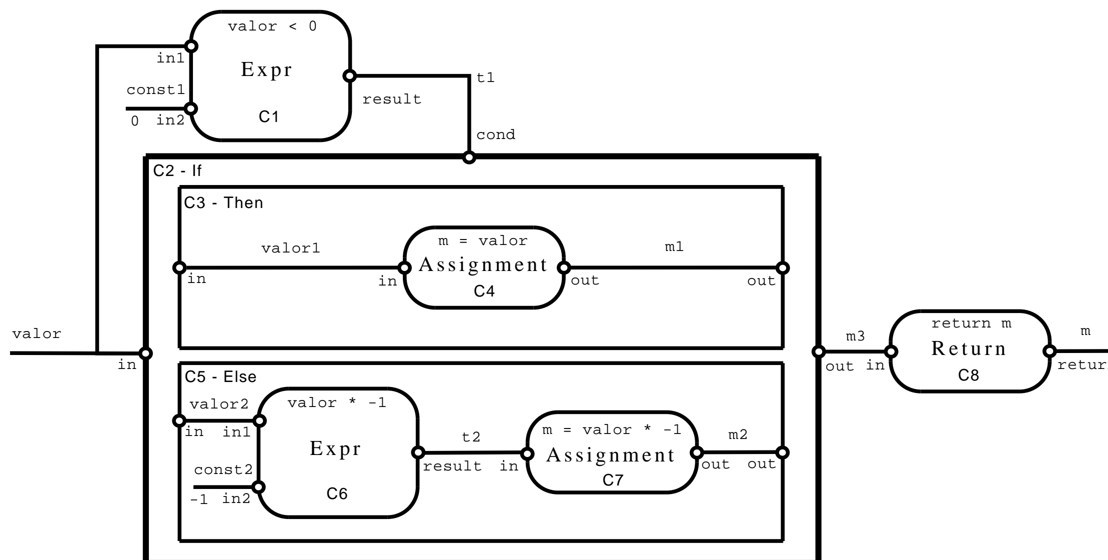


Figura 5.3: Representação do modelo estrutural do método da Figura 5.2. Cada componente (representado pelos retângulos) traz as informações: a sentença ou expressão do método que o componente representa, o tipo do componente e o seu nome.

Cada variável do método é mapeada para uma conexão. Note que o componente C_7, do tipo *Assignment*, é responsável por alterar o valor da variável m. Nesse caso, uma nova variável, m_2, é usada para representar o novo valor de m após C_7 ser executado.

A Tabela 5.3 apresenta os axiomas (fatos) utilizados para descrever o modelo estrutural do método `modulo`. Essas sentenças definem como os componentes estão conectados em função das variáveis definidas nas conexões. Por exemplo, o Axioma (A5.15) define que a porta de saída *result* do componente C_6 está relacionada com a variável t_2, enquanto o Axioma (A5.16) define que a porta de entrada do componente C_7 também está relacionada com a variável t_2. Dessa forma, é possível inferir que a conexão t_2 é usada para conectar os componentes C_6 e C_7. Também são apresentados nessa tabela os Axiomas (A5.20) e (A5.21), que definem valores para as conexões representando as constantes do método.

$in_1(C_1) = valor$	(A5.1)
$in_2(C_1) = const_1$	(A5.2)
$t_1 = result(C_1)$	(A5.3)
$cond(C_2) = t_1$	(A5.4)
$in(C_2) = valor$	(A5.5)
$out(C_2) = m_3$	(A5.6)
$in(C_3) = valor_1$	(A5.7)
$out(C_3) = m_1$	(A5.8)
$in(C_4) = valor_1$	(A5.9)
$out(C_4) = m_1$	(A5.10)
$in(C_5) = valor_2$	(A5.11)
$out(C_5) = m_2$	(A5.12)
$in_1(C_6) = valor_2$	(A5.13)
$in_2(C_6) = const_2$	(A5.14)
$result(C_6) = t_2$	(A5.15)
$in(C_7) = t_2$	(A5.16)
$out(C_7) = m_2$	(A5.17)
$in(C_8) = m_3$	(A5.18)
$return(C_8) = m$	(A5.19)
$const_1 = 0$	(A5.20)
$const_2 = -1$	(A5.21)

Tabela 5.3: Sentenças do modelo estrutural para o método da Figura 5.2.

Parte das sentenças[1] que compõem o modelo comportamental dos componentes apresentados na Figura 5.3 são mostradas na Tabela 5.4. Por exemplo, o Axioma (A5.23) define que as entradas dos componentes C_2 e C_3 devem ser iguais quando a condição utilizada no componente de seleção for avaliada como verdadeira ($cond(C_2) = true$).

Além dos modelos estrutural e comportamental, as entradas e saídas do caso de teste 2 são definidas como observações, e representadas pelos axiomas descritos na Tabela 5.5.

[1] Somente as sentenças necessárias para mostrar o exemplo dessa seção foram descritas. Nesse caso, somente os axiomas de simulação.

$ok(C_1) \rightarrow [result(C_1) = true \leftrightarrow in_1(C_1) < in_2(C_1)]$	(A5.22)
$ok(C_2) \wedge cond(C_2) = true \rightarrow in(C_3) = in(C_2)$	(A5.23)
$ok(C_2) \wedge cond(C_2) = true \rightarrow out(C_3) = out(C_2)$	(A5.24)
$ok(C_2) \wedge cond(C_2) = false \rightarrow in(C_5) = in(C_2)$	(A5.25)
$ok(C_2) \wedge cond(C_2) = false \rightarrow out(C_5) = out(C_2)$	(A5.26)
$ok(C_4) \rightarrow out(C_4) = in(C_4)$	(A5.27)
$ok(C_6) \rightarrow mult(in_1(C_6), in_2(C_6), result(C_6))$	(A5.28)
$ok(C_7) \rightarrow out(C_7) = in(C_7)$	(A5.29)
$ok(C_8) \rightarrow return(C_8) = in(C_8)$	(A5.30)

Tabela 5.4: Sentenças do modelo comportamental para o método da Figura 5.2. O predicado $mult(in_1, in_2, out)$ representa a operação binária $out = in_1 * in_2$.

$valor = 5$	(A5.31)
$m = 5$	(A5.32)

Tabela 5.5: Sentenças descrevendo as observações no método, definidas pelo caso de teste 2 (Tabela 5.2).

5.5.3 Geração de hipóteses

Seja SD o modelo descrevendo o método `modulo`, composto dos axiomas das Tabelas 5.3 e 5.4, $COMP = \{C_1, C_2, \ldots, C_8\}$ é conjunto de componentes existentes em SD e OBS formado pelos axiomas da Tabela 5.5. $(SD, COMP, OBS)$ é um problema de diagnóstico utilizado para encontrar as falhas no método `modulo`.

Como foram detectadas falhas no método, a fórmula:

$$SD \cup OBS \cup \{ok(C) | C \in COMP\} \tag{5.1}$$

é inconsistente.

O Algoritmo de Reiter será utilizado para encontrar as falhas em $(SD, COMP, OBS)$ e os conjuntos de contribuintes serão obtidos com o auxílio da função TP (Seção 3.6). A função TP chamará um provador de teoremas, para o qual faremos uma simulação dos passos de inferência executados para derivar uma inconsistência.

No início do Algoritmo de Reiter, a função TP é chamada para obter o conjunto de contribuintes que rotularão a raiz do grafo. A função TP chama o provador de teoremas para verificar a inconsistência da Fórmula 5.1. A Tabela 5.6 apresenta a simulação de uma possível sequência de passos de inferência (prova) feitas por um provador de teoremas.

A partir dessa prova, são verificados quais os axiomas com o predicado *ok* foram usados para extrair o conjunto de contribuintes. Nesse caso, o conjunto de contribuintes obtido é: $\{C_1, C_2, C_6, C_7, C_8\}$. Esse conjunto é utilizado para rotular a raiz do grafo sendo gerado pelo Algoritmo de Reiter.

Axiomas usados	Axioma	
de (A5.31)	$valor = 5$	
de (A5.20)	$const_1 = 0$	
de (A5.1) e (A5.31)	$in_1(C_1) = 5$	(A5.33)
de (A5.2) e (A5.20)	$in_2(C_1) = 0$	(A5.34)
de $ok(C_1)$, (A5.22) e (A5.33)	$result(C_1) = false$	(A5.35)
de (A5.3) e (A5.35)	$t_1 = false$	(A5.36)
de (A5.4) e (A5.36)	$cond(C_2) = false$	(A5.37)
de (A5.5) e (A5.31)	$in(C_2) = 5$	(A5.38)
de $ok(C_2)$, (A5.36) e (A5.25)	$in(C_5) = 5$	(A5.39)
de (A5.21)	$cont_2 = -1$	
de (A5.11) e (A5.39)	$valor_2 = 5$	(A5.40)
de (A5.13) e (A5.40)	$in_1(C_6) = 5$	(A5.41)
de (A5.14) e (A5.21)	$in_2(C_6) = 5$	(A5.42)
de $ok(C_6)$, (A5.28), (A5.41) e (A5.42)	$result(C_6) = -5$	(A5.43)
de (A5.15) e (A5.43)	$t_2 = -5$	(A5.44)
de (A5.16) e (A5.43)	$in(C_7) = -5$	(A5.45)
de $ok(C_7)$, (A5.29) e (A5.45)	$out(C_7) = -5$	(A5.46)
de (A5.17) e (A5.46)	$m_2 = -5$	(A5.47)
de (A5.12) e (A5.47)	$out(C_5) = -5$	(A5.48)
de $ok(C_2)$, (A5.26), (A5.36) e (A5.48)	$out(C_2) = -5$	(A5.49)
de (A5.6) e (A5.49)	$m_3 = -5$	(A5.50)
de (A5.18) e (A5.50)	$in(C_8) = -5$	(A5.51)
de $ok(C_8)$, (A5.30) e (A5.51)	$return(C_8) = -5$	(A5.52)
de (A5.19) e (A5.52)	$m = -5$	(A5.53)
de (A5.32)	$m = 5$	(A5.54)
de (A5.53) e (A5.54)	$\perp$	

Tabela 5.6: Sequência de passos de inferência utilizados para derivar a inconsistência da Fórmula 5.1. A coluna "Axiomas usados" mostra os axiomas usados durante a prova e a coluna "Axioma" é o resultado de uma inferência ou o próprio axioma mencionado na coluna da esquerda (quando somente um é citado nessa coluna).

A algoritmo continua a execução, expandindo o grafo. Considerando a suspensão de restrições de qualquer subconjunto de componentes do conjunto que rotula a raiz do grafo, a função TP não devolve mais nenhum conjunto de contribuintes, gerando as hipóteses de falha: $\{\{C_1\}, \{C_2\}, \{C_6\}, \{C_7\}, \{C_8\}\}$.

Observe que todo componente C_i do modelo está associado a uma sentença ou expressão do programa em uma linha L_{C_i}. Dessa forma, substituindo cada componente C_j nas hipóteses de falha pela linha L_{C_j}, são obtidas as linhas do programa possivelmente falhas: $\{\{4\}, \{7\}, \{10\}\}$. Como a Linha $\{10\}$ contém somente a instrução `return m`, as hipóteses de falha contendo essa linha podem ser eliminadas. Assim, o conjunto de hipóteses de falha é: $\{\{4\}, \{7\}\}$.

5.6 Considerações finais

Nessa seção apresentamos os conceitos de depuração de programas baseada em modelos, uma técnica derivada do diagnóstico baseado em modelos para encontrar falhas em programas. Além disso, apresentamos o conceito de modelo baseado em valor, que é uma forma possível de representar um programa.

No próximo capítulo, vamos falar sobre a Depuração Hierárquica de Programas, uma técnica que permite fazer a depuração do programa em diversos níveis de abstração, e considerando componentes abstratos, tais como: padrões elementares, função e métodos.

Parte II

Dr. Java Pro: um ambiente para introdução à programação com depuração hierárquica de programas

Capítulo 6

Depuração Hierárquica de Programas

Neste capítulo, apresentamos uma nova abordagem para fazer a depuração de programas que acreditamos ser capaz de superar algumas limitações da depuração de programas baseada em modelos tradicional (MBSD), quando aplicada em um processo de aprendizagem de programação. Nessa nova abordagem, estendemos a técnica de MBSD para utilizar o diagnóstico hierárquico baseado em modelos (HMBD), permitindo que a depuração do programa seja feita em múltiplos níveis de abstração.

6.1 Limitações do MBSD no processo de aprendizagem de programação

A técnica de depuração de programas baseada em modelos pode ser utilizada para auxiliar programadores iniciantes a aprenderem como programar [15]. Nessa abordagem, o sistema de depuração é capaz de encontrar todas as linhas do programa possivelmente falhas e apresentá-las ao aluno, que deve tomar uma ação apropriada. No processo de interação do aluno com o sistema MBSD, na tarefa de discriminação de hipóteses, o aluno deve fornecer valores para as variáveis em certos pontos do programa e, com essa interação espera-se que o aluno reconheça as falhas e aprenda com esse processo.

Um tipo de informação que a técnica MBSD não permite explorar é o conhecimento do aluno em relação aos padrões elementares e outros tipo de estruturas existentes na linguagem, tais como procedimentos e funções. Por exemplo, se um aluno entende como funciona um padrão elementar do tipo repetição contada, é interessante que o sistema de depuração automática reconheça esse padrão e use-o durante a discriminação de hipóteses para fazer a comunicação com o aluno. Assim, uma vez que o aluno aprende a construir seus programas utilizando padrões e a comunicação com o sistema de depuração é feita em termos desses padrões, uma linguagem de alto nível para fazer interação fica definida (uma linguagem em termos de estratégia de resolução de problemas).

Inspirado na técnica de HMBD, desenvolvemos uma abordagem que estende a técnica de MSBD e permite fazer a depuração hierárquica de programas, chamada de *Diagnóstico Hierárquico de Programas (Hierarchical Program Debugging* - HPD), e permite usar diversos níveis de abstração

do programa para fazer a depuração. Com essa nova abordagem, acreditamos seja possível superar algumas das limitações da técnica de MBSD quando aplicadas ao processo de aprendizagem de programação, a saber:

1. *Mesmo para programas relativamente pequenos, podem ser encontradas muitas hipóteses de falha que serão informadas ao aluno, o que pode confundi-lo, ao invés de ajudá-lo, na correção do programa, e consequentemente, no aprendizado.* Com o HPD é apresentado um conjunto menor de hipóteses de falha ao aluno e, caso ele solicite, esse conjunto poderá ser detalhado.

2. *Mostrando apenas as linhas do programa com falha traz pouca ou nenhuma informação que possa auxiliar o aluno a encontrar os erros no programa, prejudicando o aprendizado.* Nesse caso, somente uma linha do programa (contento uma ou mais instruções) é evidenciada, retirando o foco do contexto no qual ela pertence. Por exemplo, uma instrução que incrementa o valor de uma variável no corpo de um laço tem um significado maior quando é compreendido que essa instrução incrementa a variável de controle do laço, isto é, a instrução de incremento passa a ter um sentido maior no contexto ao qual ela pertence. No HPD, são apresentadas hipóteses relacionadas às abstrações identificadas no programa do aluno, que são mais informativas do que linhas isoladas do programa.

3. *Em muitas situações, um aprendiz de programação não consegue expressar suas intenções ou fornecer informações detalhadas sobre o comportamento do programa.* Esse problema afeta, particularmente, a sub-tarefa de discriminação de hipóteses, que requer que o aluno entenda com maiores detalhes o funcionamento de certas variáveis de programa (relacionadas às hipóteses de falha apresentadas pelo sistema de depuração automática). Com o uso de HPD, espera-se que o aluno possa expressar suas intenções relacionadas às abstrações que foram identificadas no seu programa.

Na seção a seguir discutimos os tipos possíveis de abstrações e como elas podem ser utilizadas na depuração de programas.

6.2 Depuração de programas com abstrações

Diversos tipos abstrações podem ser utilizadas no processo de construção de programas. Sempre que for possível agrupar um conjunto de linhas de forma a compor uma unidade que desempenhe uma função específica dentro do programa, podemos caracterizar uma abstração. Assim, estruturas como: laços, condicionais, funções, procedimentos e padrões elementares podem ser considerados abstrações. Nesse trabalho, consideramos como abstrações somente alguns padrões elementares, laços, seleções, procedimentos e funções.

Uma abordagem para fazer a depuração hierárquica de programas consiste em tratar as abstrações citadas como componentes abstratos e fazer com que o sistema de depuração automática

de programas entenda esses componentes abstratos, da mesma forma como é feito no HMBD, mas adaptando o algoritmo para as nossas necessidades pedagógicas.

O uso de componentes abstratos em um sistema de depuração automática pode trazer as seguintes vantagens:

- estabelecer o diálogo com o aprendiz em termos de estratégias de resolução de problemas, i.e., através de uma linguagem de comunicação de alto nível. Isso se de deve ao fato de cada componente abstrato ter uma semântica definida e o seu comportamento ser conhecido pelo aluno. Por exemplo, suponha que um aluno tenha criado um método para calcular fatorial de um número inteiro. Ao fazer um chamada desse método, o aluno não precisa conhecer os detalhes da implementação para saber o comportamento dele, simplesmente sabendo que o método calcula o número fatorial, o aluno é capaz de predizer qual o valor que deve ser devolvido pelo método para uma dada entrada. Utilizando a abordagem de depuração hierárquica, o aluno tem a possibilidade de visualizar essa chamada para o método que calcula o fatorial como sendo uma abstração ou então optar por ver os detalhes dessa chamada, e consequentemente os detalhes da implementação do método;

- permitir o raciocínio sobre o programa de uma forma hierárquica, i.e., detectar falhas em vários níveis de abstração. Nesse caso, o aluno decide qual o nível de abstração do programa que ele deseja fazer a depuração, que pode considerar os componentes abstratos ou uma visão mais detalhada contendo os componentes internos de alguns componentes, podendo chegar até um nível sem componentes abstratos, considerando somente as linhas do programa.

Note que, para garantir que essa abordagem de depuração proposta seja efetiva na aprendizagem, é necessário que os alunos que venhan a utilizar a ferramenta possuam algum conhecimento prévio sobre abstrações em programação. Assim, é interessante que esses tenham cursado uma disciplina introdutória de programação que forneça essa informação sobre abstrações ou que siga uma metodologia de ensino/aprendizagem auxiliada por padrões elementares.

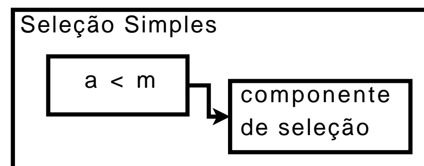
6.3 Representação de abstrações programas

Para cada tipo de abstração que será reconhecida pelo depurador automático, deve haver um componente abstrato que será utilizado para representá-la. Esse componente abstrato deve ser capaz de representar o comportamento das sentenças individuais do programa e de outras abstrações, de forma a manter o comportamento e a estrutura do modelo gerado equivalente ao modelo sem componentes abstratos, como é feito no MBSD tradicional.

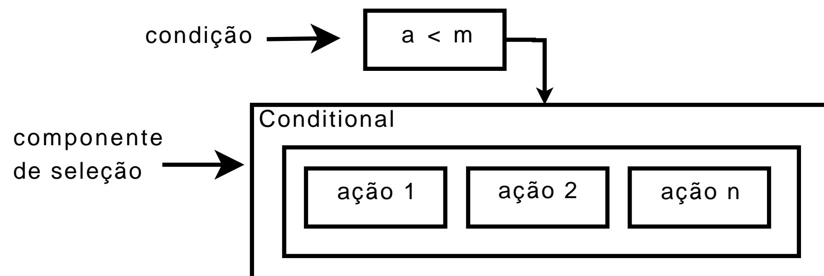
Analogamente ao que foi apresentado no Capítulo 4, um componente abstrato que representa uma abstração em um programa é composto por: modelo estrutural interno, modelo estrutural externo e modelo comportamental, construídos a partir das informações dos modelos de seus com-

ponentes internos. Nesse caso, os componentes internos podem ser outros componentes abstratos ou componentes que representam sentenças do programa. A Figura 6.1, mostra como é gerado um componente abstrato que representa o padrão elementar de Seleção Simples a partir do programa do aluno. Na Figura 6.1(c), é apresentado o trecho de um programa no qual foi aplicado o padrão elementar de Seleção Simples. A Figura 6.1(b), mostra os componentes gerados no nível base para o programa. Por fim, na Figura 6.1(a), os componentes do nível base foram agregados num componente abstrato que representa o padrão de Seleção Simples.

a) Modelo com o componente representando
o padrão de Seleção Simples



b) Modelo base gerado para o
trecho de programa apresentado



c) Padrão de Seleção Simples
aplicado usada em um programa

```
if (a < m) {
  ação 1;
  ação 2;
  ...
  ação n;
}
```

Figura 6.1: Etapas na construção de um componente representando o padrão elementar de Seleção Simples.

Na abordagem de depuração hierárquica baseada em modelos tradicional, HMBD, cada componente abstrato tem seu comportamento definido de forma prévia, isto é, um componente abstrato do tipo `NAND` sempre possui como componentes internos um componente `AND` e um inversor. No caso da representação de componentes abstratos de programas, essa informação não está previamente definida por completo. Por exemplo, um componente abstrato que representa um padrão elementar de Repetição Contada (Apêndice A) deve ser capaz de representar o comportamento de um laço e de todas as sentenças no corpo do laço, que podem ser quaisquer sentenças válidas na linguagem.

Assim, o componente abstrato deve ser flexível o suficiente para representar o comportamento dessas construções. Utilizaremos um exemplo para mostrar como são construídos os modelos abstratos para o programa de um aluno.

Exemplo 6.1 - Problema da parcela. O valor de uma parcela a ser paga é obtida aplicando as duas seguintes regras:

1. caso o pagamento seja feito em atraso, deverá haver um aumento na parcela de 50, caso contrário, deverá ser aplicado um desconto de 25;
2. caso o histórico do cliente indique que ele é um mal pagador, o valor da parcela deverá ser incrementada em 15, caso contrário, deverá have um desconto de 5 na parcela.

Construa um programa que recebe como entrada três número inteiros, (1) o valor da parcela; (2) um inteiro 0 ou 1 indicando que o pagamento foi feito em dia ou em atraso, respectivamete e; (3) um inteiro 0 ou 1 indicando se o cliente tem um histórico de mal pagador ou não; e imprima o valor final para a parcela. □

O Programa 3 foi construído por um aluno para resolver o problema do Exemplo 6.1. O método `readInt()` foi usado para obter um número inteiro da entrada padrão e `writeInt` para imprimir um valor inteiro na saída. Observe que o aluno aplicou o padrão de Seleção Alternativa nesse programa, nas linhas 7 e 10. Apesar do aluno ter implementado a estratégia correta no programa, ele cometeu uma falha na linha 10, usando o operador "==" invés de "! =".

Programa 3 Programa construído por um aluno para solucionar o Problema da Parcela.

```
    1   public static void main(String[] args) {
    2     int , parcela, atraso, malPagador;
    3     parcela = readInt();
    4     atraso = readInt();
    5     malPagador = readInt();
    6     valorTotal = parcela;
+   7     if (atraso == 1)
+   8       valorTotal = valorTotal + 50;
+   9     else valorTotal = valorTotal -25;
*  10     if (malPagador == 1)
*  11       valorTotal = valorTotal - 5;
*  12     else valorTotal = valorTotal + 15;
   13     writeInt(valorTotal);
   14   }
```

A partir do Programa 3, é construído o modelo baseado em valor de nível 0 (modelo base) e, a partir desse modelo, é construído um modelo abstrato. A Figura 6.2 mostra esses dois modelos. O modelo abstrato da Figura 6.2(a) possui os componentes abstratos $CA1$ e $CA2$ que representam

os padrões elementares identificados no programa (Seleção Alternativa nas linhas 7 e 10). Esses componentes abstratos têm como componentes internos $\{C4, C5\}$ e $\{C10, C11\}$, respectivamente. Para cada componente abstrato é necessário especificar os modelos estruturais interno e externo, e o modelo comportamental. A Tabela 6.1 mostra esses modelos para o componente $CA1$.

Modelo estrutural interno

$composit(C, E, Cond) \rightarrow$
$in_2(C) = in_1(E) \wedge in_3(C) = in_2(E) \wedge$
$in_1(C) = in_1(Cond) \wedge result(E) = aux4 \wedge$
$out_1(C) = out_1(Cond) \wedge condResult(C) = aux4$
$composit(CA1, C4, C5)$

Modelo comportamental

$cond(C) \wedge ok(C) \rightarrow \exists E, Cond\, [$
$cond - composit(C, E, Cond) \wedge$
$ok(E) \wedge ok(C)\ ]$
$cond(CA1)$

Modelo estrutural externo

$in_1(CA1) = valorTotal3$
$in_2(CA1) = atraso1$
$in_3(CA1) = const1$
$out_1(CA1) = valorTotal5$

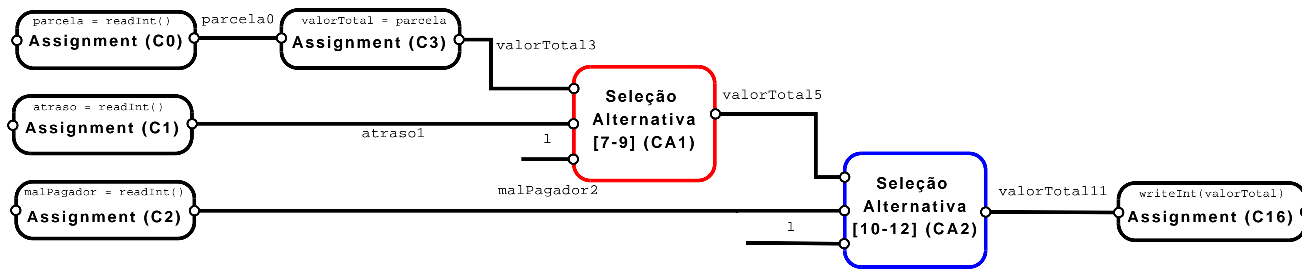
Tabela 6.1: Modelos descrevendo o componente abstrato $CA1$.

6.4 Discriminação de Hipóteses

Ao contrário do que é feito no HMBD, que a discriminação de hipóteses ocorre somente após terem sido encontradas as hipóteses de falha no modelo base, para fazer depuração de programas consideramos conveniente que a tarefa de discriminação de hipóteses seja feita para cada nível de abstração.

A ideia principal para se utilizar abstrações em sistemas físicos é para reduzir o esforço computacional para encontrar os componentes falhos. No caso dessa extensão do HMBD para fazer depuração de programas, devemos ter em mente que cada componente abstrato tem um comportamento alto nível que o aluno conhece. Dessa forma, é possível utilizar o componente abstrato para comunicar a falha ao aluno em uma linguagem de alto nível (em termos de padrões, no caso), dando a possibilidade dele confrontar essa informação com o conhecimento prévio adquirido nas aulas teóricas para tentar reconhecer a falha e aprender com esse processo.

Em nossa proposta, a tarefa de discriminação de hipóteses é feita através de duas operações chamadas de: *informar valores para variáveis* e *refinamento de componente abstrato*. Mas antes de especificar essas operações, definiremos os conceitos de *variável relacionada a uma conexão* e



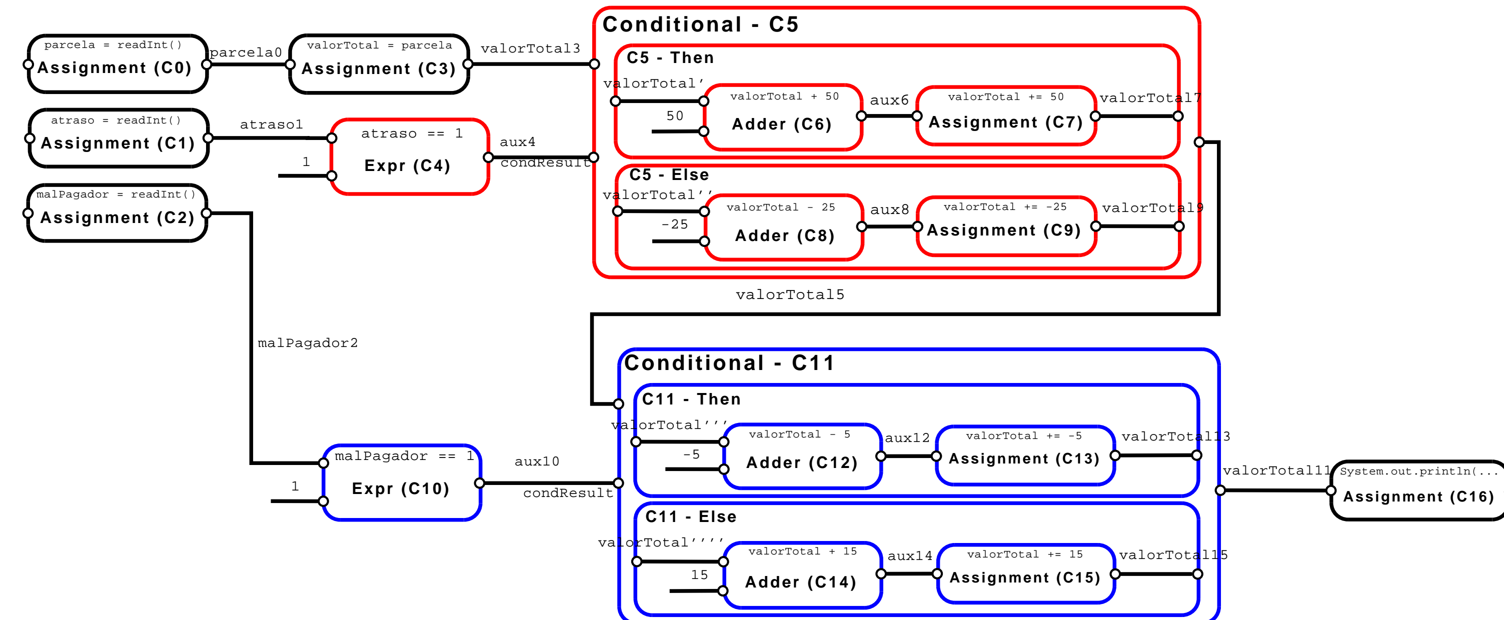


Figura 6.2: Modelos base e abstrato (nível 1) para o Programa 3.

variáveis relacionadas a uma hipótese, apresentados a seguir.

Definição 6.1 (Variável relacionada a uma conexão)**.** *Seja C um componente de um modelo baseado em valor, abstrato ou não, e conn uma conexão de entrada ou saída de C. A variável relacionada à conexão conn é a variável do programa na qual a conexão conn representa no modelo, junto da conexão conn. Uma variável v relacionada a uma conexão conn é denotada pela dupla: $\langle v, conn\rangle$.*

Definição 6.2 (Variáveis relacionadas a uma hipótese)**.** *Dado uma hipótese de falha H, composta dos componentes $\{H_1, H_2, \cdots, H_n\}$, abstratos ou não, as variáveis relacionadas à hipótese de falha H são dadas pela união de todas as variáveis relacionadas às conexões de entrada e saída dos componentes $H_i \in H$, $1 \leq i \leq n$.*

Definição 6.3 (Informar valores para variáveis)**.** *Dada uma hipótese de falha H, uma operação de informar valores para variáveis consiste em atribuir valores às variáveis relacionadas à hipótese de falha H.*

Definição 6.4 (Refinamento de componente abstrato)**.** *Dada uma hipótese de falha H, a operação de refinamento consiste em aplicar o Axioma 4.1 a um ou mais componentes abstratos de H.*

Na discriminação de hipóteses, quando o valor é informado para uma variável é necessário saber exatamente em qual conexão esse valor será atribuído, visto que diversas conexões estão relacionadas a uma mesma variável, mas em diferentes pontos do programa (essa é uma característica do modelo baseado em valor, apresentado no Capítulo 5). Por essa razão as Definições 6.1 e 6.2 são necessárias.

Na operação de informar valores para variáveis, o valor informado pelo aluno é comparado com o valor gerado pelo depurador. Caso esses valores sejam iguais, então o programa está fazendo o que o aluno queria (i.e., de acordo com as intenções do aluno), ou seja, a falha não está naquele ponto do programa e o aluno deve tentar discriminar outra hipótese de falha. Nesse momento, o depurador automático remove essa hipótese de falha e o aluno deve escolher outra hipótese para discriminar. Por outro lado, caso o valor informado pelo aluno seja diferente do valor gerado pelo depurador, então pode ser que aquela falha seja realmente a falha real no programa e o aluno deve alterar o programa tentando corrigir aquela falha. Note que não é possível do depurador descobrir que a falha real está em uma linha (ou um conjunto de linhas) específica do programa, a não ser que seja feito todo o processo de discriminação de hipóteses e, ao final fique somente uma hipótese de falha. Isso acontece porque mesmo que seja reconhecida a falha em uma linha k do programa, essa falha pode ser decorrente da execução de outra instrução com falha que gerou valores inválidos, e que foram usados pela instrução na linha k.

No caso da operação de refinamento de componente abstrato, o aluno tem a possibilidade de vizualizar os detalhes do componente abstrato no qual ele refinou. Nesse caso, o depurador automático gera novamente as hipóteses de falha considerando os componentes internos desse componente abstrato.

O Exemplo 6.2 mostra como as operações de informar valores para variáveis e refinamento de componente abstrato são aplicadas.

Exemplo 6.2. Seja $H = \{SelecaoAlternativa[7-12]\}$ uma hipótese de falha gerada utilizando o modelo abstrato da Figura 6.2(a), gerado para o Programa 3. Essa hipótese de falha informa que o padrão de Seleção Alternativa na linha 7 está com falha.

Ao executar a operação de informar valores para as variáveis de H, é possível informar valor para as seguintes variáveis:

- $valorTotal$ e $atraso$, que são representadas pelas conexões $valorTotal3$ e $atraso1$, respectivamente, e são entradas para o componente abstrato representando o padrão elementar de Seção Alternativa;
- $valorTotal$, representado pela conexão $valorTotal5$, que é o novo valor da variável $valorTotal$ que foi modificada na Seleção Alternativa.

Ainda com a hipótese H, é possível refinar o componente abstrato $SelecaoAlternativa[7-12]$. A operação de refinamento faz com que o componente abstrato $SelecaoAlternativa[7-12]$ seja substituído pelos seus componentes internos $C4$ e $C5$, e o depurador automático passa a considerar esses componentes na geração de hipóteses. □

É importante notar que, ao definirmos a operação de refinamento de componente abstrato, o conceito de níveis de abstração definido para o HMBD tradicional deixa de ser válido. Isso porque se um modelo abstrato em um nível $k > 0$, define pelo menos 2 componentes abstratos que possuem seus componentes internos no nível $k-1$, é possível refinar somente um desses componentes, gerando um modelo intermediário diferente dos modelos nos níveis k e $k-1$. No caso do HMBD, isso é impossível de acontecer pois o Algoritmo 4 (*Top-Down-Diagnosis*) ao aplicar as relações de comportamento (Definição 4.3), simplesmente muda para um modelo mais detalhado. Note que, apesar dessa diferença, a construção do modelo e dos componentes abstratos pode permanecer a mesma. Em nossa implementação, essa diferença citada é resolvida pelo próprio mecanismo de refinamento implementado. Além disso, o termo *nível de abstração* também será utilizado para fazer referência a esses modelos em níveis de abstração intermediários.

Na seção a seguir, apresentamos o algoritmo que permite fazer a depuração hierárquica de programas.

6.5 O algoritmo HPD

Como citado anteriormente, em nossa abordagem é possível que exista a interação do aluno em cada nível de abstração. Assim, as hipóteses de falha são apresentadas ao aluno logo após serem encontradas em cada nível de abstração e, para cada hipótese o aluno poderá:

1. *Reconhecer que aquela hipótese corresponde de fato ao seu erro, modificando em seguida o programa que será novamente verificado com os casos de teste.* Caso o programa modificado ainda apresente falhas, o aluno pode reiniciar o processo de depuração no programa modificado ou voltar à versão anterior do programa e escolher outra hipótese de falha para continuar com a depuração.

2. *Pedir que o sistema de depuração o ajude a verificar se a hipótese de falha em questão corresponde de fato ao erro do programa.* Nesse caso, o aluno deve fornecer informações adicionais a respeito de suas intenções (informar valores para variáveis), e.g.: o valor esperado para uma variável durante a execução do programa. Caso a intenção do aluno corrobore com as predições do sistema de depuração, a hipótese de falha será eliminada e um outro conjunto de hipóteses de falha será apresentado. Caso contrário, recaímos no caso 1.

3. *Optar por fazer um diagnóstico mais detalhado, ao detectar um componente abstrato com uma possível falha.* Para isso, o aluno pode requerer que o sistema substitua um componente abstrato pelos seus componentes internos (refinamento de componente abstrato), para que seja feito o diagnóstico em um modelo mais detalhado. Nesse caso, um novo conjunto de hipóteses de falha é gerado.

O Algoritmo 7, mostra como é feita a depuração hierárquica de programas, considerando esses passos de interação citados.

6.6 Exemplo de depuração com o HPD

Nessa seção, apresentamos um exemplo da execução do Algoritmo 7 para encontrar as falhas no Programa 3. No Capítulo 7, mostraremos também um exemplo utilizando a ferramenta Dr. Java Pro, na qual disponibilizamos o nosso depurador automático.

Seja CT um caso de teste para o Problema da Parcela, apresentado no Exemplo 6.1, com as entradas: 1000, 0 e 0; e saída esperada igual a 970. Executando o Programa 3 com as entradas definidas no casos de teste TC, o programa devolve o valor 990 ao invés de 970 (saída esperada por TC), indicando que há falhas no programa.

Utilizando a técnica MBSD para fazer a depuração do programa, são obtidas a seguintes hipóteses de linhas com falha: $\{\{12\}, \{9\}, \{6\}, \{10\}, \{7\}\}$. Note que, a linha 10 contendo a falha funcional do programa está entre as hipóteses de falha. No caso da abordagem HPD, Algoritmo 7, é necessário informar os seguintes parâmetros:

- o programa do aluno;
- os padrões elementares utilizados no programa, nesse caso o padrão de Seleção Alternativa aplicado nas linhas 7 e 10;

Algoritmo 7: `HPD(`P`, `A`, `TC`)`

Entrada: P: programa do aluno;
A: conjunto de abstrações identificadas em P;
TC: caso de teste para o programa P.

1 construir os modelos para o programa P, considerando as abstrações no conjunto A;
2 encontrar as hipóteses de falha do programa utilizando o modelo mais abstrato (M') ;
3 **enquanto** *houver alguma hipótese de falha no programa, considerando o modelo* M' **faça**
4 apresente as hipóteses de falha para o aluno e aguardar pela sua ação;
5 **se** *o aluno optou pela ajuda do sistema para verificar uma hipótese escolhida* **então**
6 permita ao aluno informar valores para as variáveis relacionadas à hipótese escolhida ;
7 **se** *as novas observações são discrepantes das predições* **então**
8 o aluno pode ter encontrado a(s) falha(s) e ele deverá modificar o programa ;
9 **senão**
▷ `o aluno deve continuar a discriminação de hipóteses com as observações atuais`
10 encontrar as novas hipóteses de falha do programa considerando o modelo M' e as novas observações ;
11 **se** *o aluno optou por modificar o programa* **então**
12 finalize o processo de depuração ;
13 volte ao modo de edição do programa ;
14 **senão se** *o aluno optou por refinar um conjunto de componentes abstratos* CA *pertencentes à hipótese escolhida* **então**
▷ M' `conterá os componentes internos dos componentes abstratos em` CA.
15 substitua em M' os modelos dos componentes em CA pelos modelos de seus respectivos componentes internos ;
16 encontre as novas hipóteses de falha do programa considerando o modelo M' que agora contém os componentes internos dos componentes CA ;
17 **fim**

- o caso de teste TC.

O processo de depuração inicia com a construção dos modelos base e abstrato (Figura 6.2), no qual são usadas as definições das abstrações informadas para gerar os componentes abstratos. Em seguida, o depurador gera as hipóteses de falha utilizando o modelo mais abstrato (Figura 6.2)), e devolve as seguintes hipóteses de falha:

$$\Delta_1 = \{\{6\}, \{SelecaoAlternativa[7-9]\}, \{SelecaoAlternativa[10-12]\}\}$$

Observe que as hipóteses de falha devolvidas em Δ_1 trazem linhas isoladas do programa, por exemplo, a hipótese $\{6\}$, e também componentes abstratos, como é o caso da hipótese: $\{SelecaoAlternativa[7-9]\}$, informando que a falha pode estar no componente abstrato representando o padrão elementar de Seleção Alternativa entre as linhas entre 7 e 9.

As hipóteses de falha em Δ_1 são comunicadas ao aluno, que deverá tomar uma ação de acordo com os passos apresentados na Seção 6.5.

Suponha que o aluno tenha analisado seu programa e verificado que o problema está no padrão

elementar de Seleção Alternativa, na linha 10, representado pela hipótese de falha: $\{SelecaoAlternativa[10-12]\}$. Nesse caso, o aluno pode informar valores para as variáveis relacionadas a essa hipótese de falha ou então refinar o componente abstrato dessa hipótese, para verificar as falhas em um nível mais detalhado, considerando os componentes internos dos componentes abstratos refinados. Suponha que, nesse caso, o aluno opte por refinar o componente abstrato representado por $SelecaoAlternativa[10-12]$. O sistema de depuração substitui os modelos desse componente abstrato pelo modelos dos seus componentes internos e faz o diagnóstico novamente. Nesse caso, as hipóteses de falha devolvidas pelo sistema de depuração automática são:

$$\Delta_2 = \{\{6\}, \{SelecaoAlternativa[7-9]\}, \{10\}, \{12\}\}.$$

Suponha que o aluno escolha a hipótese de falha $\{6\}$ para informar valores para as variáveis. Para essa hipóteses de falha é possível informar valores para a variável $valorTotal$, entre as linhas 6 e 7 do programa. Se o aluno informar o valor 1000 para essa variável (que é o valor inicial que essa variável deve assumir para depois aplicar os descontos ou incrementos), o depurador automático devolve as seguintes hipóteses de falha:

$$\Delta_3 = \{\{SelecaoAlternativa[7-9]\}, \{10\}, \{12\}\}.$$

O aluno pode continuar informando valores para variáveis até que o depurador devolva somente uma hipótese de falha ou então, se ele conseguir identificar qual a falha real, ele pode selecionar a hipótese que representa essa falha e usar a operação de informar valores para variáveis para obter uma confirmação do depurador automático. Por exemplo, se o aluno selecionar a hipótese de falha $\{10\}$ e informar que o resultado esperado para a condição dessa seleção é *verdadeiro*, será devolvida somente a seguinte hipótese de falha:

$$\Delta_4 = \{\{10\}\}.$$

Note que, mesmo tendo devolvido uma hipótese de falha composta de uma única linha do programa, não foi necessário refinar o componente abstrato $SelecaoAlternativa[7-9]$ para chegar nesse resultado.

6.7 Considerações finais

Nesse capítulo, apresentamos uma abordagem para fazer a depuração automática de programas em múltiplos níveis de abstração. Cada nível de abstração é composto de componentes simples e abstratos, que são utilizados na geração de hipóteses de falha do depurador. As hipóteses de falha encontradas em cada nível de abstração são comunicadas ao aluno, que pode fazer a discriminação

nesses níveis abstratos, através das duas operações que definimos, a saber: informar valores para variáveis e refinamento de componente abstrato.

Além disso apresentamos um algoritmo para fazer a depuração hierárquica que considera a interação com o aluno durante o processo. Com esse algoritmo é possível superar algumas limitações do HMBD tradicional, citadas no Capítulo 4 (Seção 4.7).

Ao contrário do que é feito no HMBD tradicional, quando a discriminação de hipóteses é feita em vários níveis de abstração é possível reduzir o espaço de busca de componentes falhos nos níveis mais detalhados. Isso ocorre porque uma vez que um componente abstrato foi reconhecido como tendo o comportamento correto (não anormal) em algum nível abstrato, não é necessário considerar seus componentes internos durante a geração de hipóteses nos níveis mais detalhados. Além disso, se o aluno reconhecer que a falha está em um componente abstrato, nossa abordagem possibilita que somente esse componente seja explorado, através da operação de refinamento, até chegar na linha do programa com falha. Nesse momento, o aluno pode interromper o processo de depuração sem ter que chegar até um modelo sem componentes abstratos. Por fim, nessa abordagem não existe o problema da não diagnosticabilidade que pode ocorrer no HMBD, porque as entradas e saídas para o programa sempre estão disponíveis, independente do nível de abstração.

Essa nossa abordagem também permite superar algumas limitações da técnica MBSD quando aplicada à aprendizagem de programação. Com a técnica HPD, são consideradas abstrações do tipo: funções, métodos e padrões elementares, durante o processo de depuração. Assim, as falhas comunicadas ao aluno são em termos de componentes abstratos representando essas abstrações, formando uma linguagem de alto nível para a interação com o aluno. Além disso, ao considerar os componentes abstratos durante a depuração são apresentadas menos hipóteses de falha ao aluno, evitando que ele fique confuso com uma grande quantidade de informação, como acontece no MBSD.

No próximo capítulo, apresentamos alguns detalhes da implementação do nosso depurador automático que utiliza a técnica de depuração hierárquica de programas.

Capítulo 7

Implementação do Algoritmo HPD

Nesse capítulo, apresentamos em detalhes como foi implementado o nosso depurador automático de programas, HPD, especificado no Capítulo 6. Além disso, também apresentamos o Dr. Java Pro, um ambiente para o desenvolvimento de programas no qual disponibilizamos esse depurador automático de programas.

7.1 Implementação do depurador automático de programas

A implementação do depurador automático de programas foi dividida em subprojetos que realizam tarefas específicas, tais como: construção do modelo com abstrações, manipulação de grafos (necessárias para o algoritmo *Minimal Hitting Set*), execução de casos de teste, entre outras. A Tabela 7.1, apresenta cada um desses subprojetos e uma descrição sobre a sua finalidade.

Subprojeto	**Descrição**
jGraphLib	Biblioteca geral para trabalhar com grafos, que também disponibiliza uma implementação do algoritmo *Minimal Hitting Set* (algoritmo de Reiter).
java-compiler	Analisador sintático para o subconjunto da linguagem Java utilizada nesse trabalho, gerado pela ferramenta de construção de compiladores SableCC (`http://sablecc.org`).
program-diagnosis	Contém componentes do modelo e implementação dos algoritmos de depuração (inclusive o HPD).
J2VBModel	Gerador do modelo baseado em valor do programa Java (Seção 5.3) com múltiplos níveis de abstração.
java-debugger	Disponibiliza funcionalidades para ler as configurações de um caso de teste executar o depurador.

Tabela 7.1: Projetos utilizados no desenvolvimento do sistema de depuração automática.

A Figura 7.1 mostra um diagrama de dependência entre esses subprojetos. Cada caixa é um subprojeto e as setas mostram as suas dependências. O subprojeto a partir do qual parte uma seta depende do subprojeto no qual a seta chega.

A seguir, descrevemos os detalhes de implementação das etapas que ocorrem após a detecção de sintomas até a discriminação de hipóteses. É importante citar que a detecção de sintomas é feita

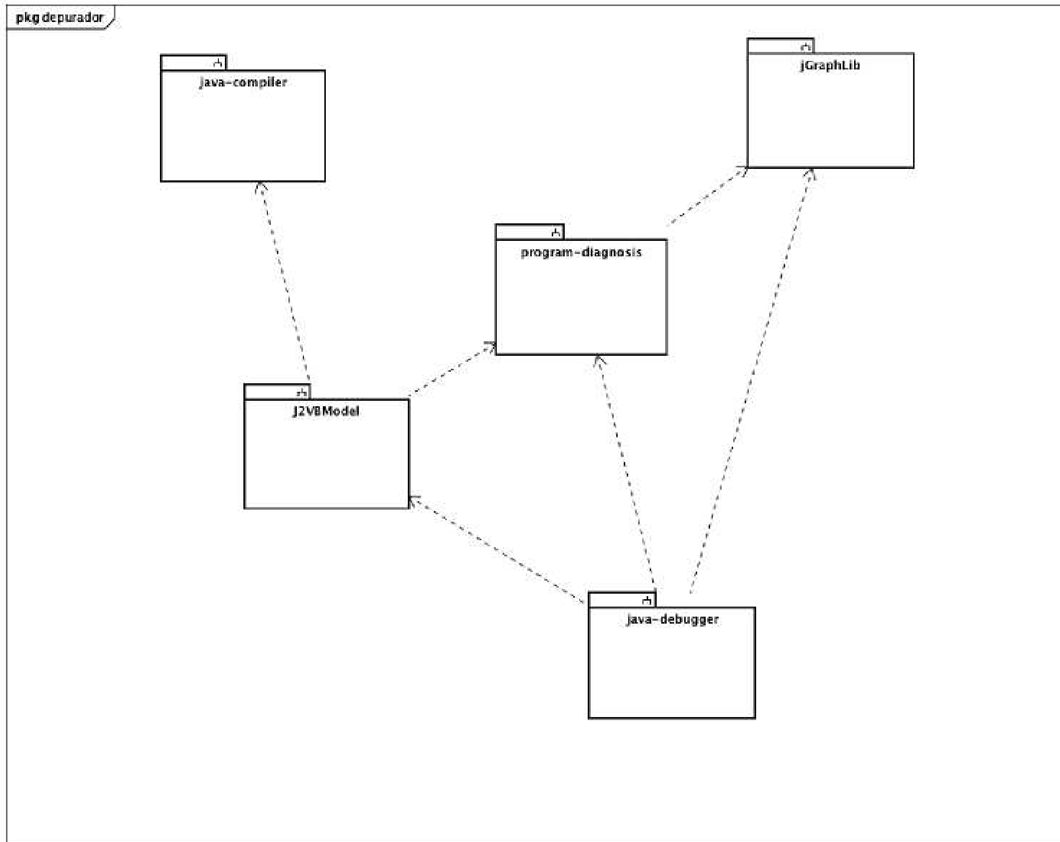


Figura 7.1: Dependência entre os projetos construídos para implementar a ferramenta de depuração automática de programas.

em uma etapa preliminar, através da execução dos casos de teste para o programa do aluno.

7.1.1 Análise sintática

A etapa de análise sintática é responsável por gerar a árvore sintática do programa, que será posteriormente interpretada para gerar o modelo baseado em valor para esse programa. A partir da gramática que compreende um subconjunto da linguagem Java (especificada no Capítulo 5), utilizamos um gerador de compiladores chamado SableCC (`http://sablecc.org`) para construir um compilador capaz de reconhecer essa linguagem. Optamos por usar o SableCC devido ao fato dele gerar um compilador orientado à objetos que disponibiliza implementações que permitem visitar a árvore sintática de uma forma simples, além de ser altamente extensível e de código aberto. É importante citar que na etapa da análise sintática espera-se que o programa do aluno não contenha erros de compilação. O subprojeto que disponibiliza o analisador sintático é o java-compiler.

7.1.2 Construção do modelo

A etapa da construção do modelo é responsável por interpretar a árvore sintática gerada na etapa de análise sintática e derivar um modelo baseado em valor para o programa do aluno. O modelo gerado pode conter componentes abstratos e deve possuir toda informação necessária para fazer a depuração em múltiplos níveis de abstração. O subprojeto responsável por fazer a construção do modelo é o J2VBModel, que utiliza as funcionalidades disponíveis no projeto java-compiler.

O Algoritmo 8 mostra como é feita a geração desse modelo do programa. Esse algoritmo recebe como entrada um programa P, um conjunto de informações que permite criar as abstrações; e devolve um modelo baseado em valor com múltiplos níveis de abstração, que representa o programa P. Nessa fase de construção do modelo, somente são construídas abstrações para representar funções (métodos que devolvem valores), procedimentos (métodos que não devolvem valores) e os padrões elementares de Seleção Simples, Seleção Alternativa e Repetição Contada. As demais abstrações devem ser construídas posteriormente, mas não cobriremos esse assunto nesse trabalho.

Algoritmo 8: $construir - VBM(P, A)$

Entrada: P: programa a partir do qual será derivado o modelo baseado em valor com abstrações,
A: informações sobre abstrações que serão utilizadas na construção do modelo.
Saída: Um modelo baseado em valor com múltiplos níveis de abstrações que representa o programa P.

1 $AST \leftarrow parse(P)$;
2 $r \leftarrow$ obter o nó raiz de AST ;
3 $MP \leftarrow$ criar uma representação do modelo do programa vazia (sem componentes e sem conexões) ;
 ▷ `constrói os componentes e as conexões de` M `com base na` AST
4 $interpretarAST(r, A, MP)$;
5 **devolva** MP ;

Algoritmo 9: $interpretarAST(N, A, S)$

Entrada: N: nó da árvore sintática do programa,
A: informações sobre abstrações que serão utilizadas na construção do modelo.
MP: modelo do programa no qual serão construídos os componentes e as conexões geradas a partir da árvore sintática, AST

1 **se** *node não for nulo* **então**
2 preProcessarNo(N, A, MP) ;
3 $Nesq \leftarrow$ nó esquerdo de r ;
4 interpretarAST($Nesq, A, MP$) ;
5 $Ndir \leftarrow$ nó direito de r ;
6 interpretarAST($Ndir, A, MP$) ;
7 posProcessarNo(N, A, MP) ;
8 **fim**

Na Linha 1 do Algoritmo 8, é feita uma chamada para a função `parse`, que é gerada automaticamente pelo SableCC, para construir a árvore sintática do programa. Em seguida, é criada uma estrutura para representar um modelo vazio do programa, que ainda não contém componentes e nem conexões (essa estrutura será posteriormente modificado para conter os componentes e as

conexões utilizadas para representar o programa no nível mais abstrato).

Com a árvore sintática do programa é possível gerar o modelo baseado em valor, utilizando o Algoritmo 9. Esse algoritmo recebe como entradas: a árvore sintática do programa, as informações sobre as abstrações e a estrutura que representa o modelo vazio do programa, e faz uma busca em profundidade na árvore sintática, visitando cada nó através das duas funções: $preProcessarNo$ e $posProcessarNo$. Essas funções são utilizadas para transformar cada nó da árvore em um ou mais componentes do modelos e fazer as conexões. Através da função $preProcessarNo$ é possível visitar cada nó da árvore em *pré-ordem*, enquanto que, através da função $posProcessarNo$ é possível visitar os nós em *pós-ordem*. Note que esse comportamento é diferente do apresentado no Algoritmo 6, na Seção 5.4, que utiliza somente uma busca em pré-ordem. Essa adaptação permite obtermos um controle melhor sobre a criação de componentes e conexões, não necessitando que sejam feitas diversas buscas em uma mesma subárvore para fazer as conexões [27]. Os componentes gerados pelas funções de visitação são inseridos no modelo do programa passado como parâmetro.

Durante essa fase de interpretação também são construídos os componentes abstratos. Ao encontrar um nó n da árvore abstrata, que representa uma abstração no programa (por exemplo, uma instrução *if-then* corresponde a um padrão elementar de Seleção Simples; um *if-then-else* corresponde a um padrão elementar de Seleção Alternativa), é criado um componente abstrato CA correspondente, mas ainda sem seus componentes internos. Em seguida, a subárvore enraizada em n é interpretada normalmente. Ao terminar a intepretação dessa subárvore, é feita uma busca nos componentes gerados para definir quais serão usados como componentes internos de CA. Note que esse é um processo recursivo, através do qual podem ser construídos componentes abstratos que possuem como componentes internos outros componentes abstratos.

Existem alguns detalhes que devem ser tratados durante a construção do modelo do programa. tais como:

- O modelo do programa é um *modelo livre de laços* (Seção 5.3.4), no qual os laços são transformados em seleções encadeadas. O número de seleções encadeadas é dado pelo número de iterações do laço, que no nosso caso, é definido como um parâmetro fixo.
- Para facilitar a geração do *modelo livre de laços*, é feito um pré processamento no qual os laços do tipo `for` são convertidos em laços do tipo `while`. Dessa forma o procedimento que faz essa conversão precisa tratar somente laços do tipo `while`.
- Todo método definido no programa é definido como um subsistema e suas entradas e saídas não são conectadas a nenhum componente. Os componentes que representam as chamadas de métodos são responsáveis por fazerem as conexões com o subsistema representando o método sendo chamado, durante a propagação de valores. Isso permite que o subsistema representando um método seja reutilizado em várias chamadas, mas impede que sejam feitas chamada recursivas.

- Para toda sentença `if` no programa que não tenha um `else`, deve ser construído um bloco `else` fictício, que no modelo será usado para gerar um componente que simplesmente transporta os valores de entrada para a saída (e vice-versa no caso da propagações de trás para frente). Esse tratamento é necessário para garantir que a propagação seja feita corretamente,

7.1.3 Transformação de entradas/saídas de um caso de teste em observações

Para executar o depurador automático é necessário fornecer um caso de teste que foi executado sem sucesso com o programa em questão. As entradas e saídas desse caso de teste devem ser usadas pelo depurador para fazer as propagações e detectar os conflitos, permitindo assim gerar as hipóteses de falha. Vamos chamar o conjunto de entradas e saídas de um caso de teste de parâmetros do caso de teste.

Após obter o caso de teste que será utilizado para fazer a depuração, seus parâmetros são usados para atribuir valores às conexões do modelo. Para cada parâmetro do caso de teste deve haver uma conexão com o mesmo nome. Por exemplo, se existe uma entrada com o nome *obs1*, deve haver no modelo uma conexão com o nome *obs1*. Assim, padronizamos a forma como são definidas as entradas e saídas para os casos de teste e também para as conexões do modelo que devem receber esses valores.

Consideramos as duas seguintes formas possíveis de um programa receber valores de entrada:

- *argumento de método*, que define um método com argumentos que serão passados na execução de um caso de teste;
- *argumentos recebidos pela entrada padrão*, que são valores obtidos através da entrada padrão do sistema operacional (por exemplo, pelo teclado).

e, as duas seguintes formas de um programa devolver valores:

- *valores devolvidos pela instrução* ***`return`***, que é o resultado devolvido na chamada de um método;
- *valores enviados para a saída padrão*, que utilizam um método específico da linguagem Java para gerar valores na saída padrão (por exemplo, o terminal do aluno).

Para as entradas do tipo *argumento de método*, são geradas no modelo tantas conexões quanto forem os argumentos do método. Cada conexão receberá um nome com o prefixo *arg*, seguido de um número sequencial, para cada argumento. No caso dos argumentos recebidos via entrada padrão, também são geradas tantas conexões quanto forem as entradas existentes no programa, e cada uma receberá um nome com o prefixo *input* seguido de um número sequencial.

Em relação às conexões representando as saídas do programa, caso seja uma conexão que represente um valor devolvido pela instrução `return`, ela possuirá o nome *argOut*. As conexões representando valores enviados para a saída padrão são construídas de forma análoga as conexões representando argumentos recebidos pela entrada padrão e recebem um nome dado pelo prefixo *output* seguido de um número sequencial.

Algoritmo 10: transformarEntradasESaidasEmObservacoesBasico(TC)

Entrada: TC: o caso de teste a partir do qual serão geradas as observações,
OBS: variável global utilizada para armazenar as observações do sistema.

1 $OBS \leftarrow \emptyset$;
2 $OBS = OBS \cup entradasdoTC$;
3 $OBS = OBS \cup saidasdoTC$;
4 **devolva** OBS ;

Para facilitar o uso das entradas e saídas de um caso de teste, utilizamos um algoritmo que recebe como entrada um caso de teste TC e disponibiliza suas entradas e saídas, com seus respectivos nomes e valores, em uma variável global que é utilizada pelo depurador. Essa variável global é chamada de observações do sistema. O Algoritmo 10 é usado para disponibilizar essas informações. O subprojeto program-diagnosis é responsável por essa implementação.

7.1.4 Propagação de restrições

A propagação de restrições é essencial para o desenvolvimento do depurador automático. Ela é responsável por propagar os valores para todos os componentes do modelo do programa, em todos os possíveis caminhos definidos pelas conexões. Uma tarefa muito importante na propagação de restrições é a detecção de conflitos, ou seja, se dois valores diferentes puderem ser gerados para uma mesma conexão, então existe um conflito envolvendo dois conjuntos de componentes em caminhos diferentes.

Algoritmo 11: propagar()

Saída: Um conjunto de componentes envolvido num conflito ou um conjunto vazio indiciando que nenhum conflito foi detectado.

1 $In1 \leftarrow$ conexão representando a entrada 1 do componente ;
2 $In2 \leftarrow$ conexão representando a entrada 2 do componente ;
3 $Out \leftarrow$ conexão representando a saída do componente ;
▷ `gera novos valores para as entradas ou para a saída, se possível.`
4 **se** *$In1$ e $In2$ possuem valores definidos* **então**
5     $Out \leftarrow In1 + In2$;
6 **senão se** *$In1$ e Out possuem valores definidos* **então**
7     $In2 \leftarrow In1 - Out$;
8 **senão se** *$In2$ e Out possuem valores definidos* **então**
9     $In1 \leftarrow In2 - Out$;
10 **se** *foi detectado algum conflito* **então**
11   **devolva** *todos os componentes envolvidos no conflito* ;
12 **devolva** *conjunto vazio* ;

Para cada componente do sistema é definida uma forma de propagar os valores, de acordo com a especificação em seu modelo comportamental (vide Capítulo 3). Na nossa implementação, cada componente do modelo é responsável por definir um método responsável por fazer propagação de valores. Por exemplo, o método de propagação para um componente que representa uma operação de adição pode ser representado pelo Algoritmo 11. Nesse algoritmo, as variáveis: $In1$, $In2$ e Out, são conexões que denotam, respectivamente, as entradas 1 e 2 do componente de adição; e a sua saída. Esse algoritmo tenta gerar um valor para a conexão de saída do componente, caso os valores de entrada estejam definidos (Linha 5), ou seja, caso o processo de propagação tenha gerado valores para as conexões relacionadas às entradas desse componente, anteriormente, esses valores podem ser usados para gerar novos valores. Caso não seja possível gerar os valores na saída, o algoritmo tentará gerar os valores nas entradas, caso a saída e pelo menos uma entrada tenham valor definido.

Caso uma conexão possua um valor x e, durante a propagação seja gerado para essa mesma conexão um valor y diferente de x, então fica caracterizado um conflito. Nesse caso, o método que está gerando os novos valores deve obter todos os componentes usados para gerar o valor x e os componentes usados para gerar o valor y, e devolvê-los ao propagador, que apontará esse conjunto como um conjunto conflito e será usado, posteriormente, para rotular um nó do grafo gerado pelo algoritmo *Minimal Hitting Set* (Capítulo 3). Caso não seja detectado nenhum conflito, o método de propagação deverá devolver um conjunto vazio.

Para que a propagação funcione corretamente no modelo do programa inteiro, é necessário que todos os componentes forneçam uma implementação para fazer a propagação, conforme o seu comportamento especificado, inclusive para os componentes abstratos.

Sempre que um novo valor é gerado na propagação, a conexão que tem esse novo valor recupera todos os componentes que estão conectados à ela e os adiciona em uma lista de componentes que poderão ser utilizados no processo de propagação para tentar gerar novos valores. Isso acontece porque, esse novo valor gerado pode ser utilizado por algum outro componente, para continuar gerando novos valores.

O Algoritmo 12, mostra como é feito o processo de propagação utilizando os componentes da lista de propagação chamada $PropagationList$, que mantém os componentes com possibilidade de gerar novos valores para conexões. Note que, para cada componente da lista é chamado o seu método de propagação, e cada uma dessas chamadas pode fazer com que novos valores sejam gerados e outros componentes sejam adicionados na lista. Dessa forma, esse algoritmo somente será finalizado quando não puder ser gerados mais nenhum novo valor para conexões do modelo.

Devido ao fato de cada componente ser responsável por fazer a sua propagação de valores, o Algoritmo 12 não precisa saber qual componente ele está solicitando que seja feita a propagação (sendo o componente abstrato ou não). Também vale citar que cada componente abstrato mantém a sua lista de propagação, que contém somente seus componentes internos. Assim, podemos considerar que existe um mecanismo de propagação "local", sendo cada componente abstrato responsável

Algoritmo 12: propogarPendentes()

```
1 enquanto PropagationList não estiver vazio faça
2     comp ← obter o primeiro componente da PropagationList ;
3     remover comp de PropagationList ;
4     comps ← chamar propagar do componente comp ;
5 fim
```

por controlar a propagação de seus componentes internos.

7.1.5 Geração das hipóteses de falha

A geração de hipóteses de falha é feita utilizando o algoritmo *Minimal Hitting Set* (vide Capítulo 3). Porém, foram feitas algumas modificações em relação ao algoritmo original, a saber:

- Ao invés de passar a tripla $\langle SD, COMP, OBS \rangle$ para o algoritmo, somente passamos uma variável que representa o modelo do programa, a partir da qual podemos obter os componentes e as conexões, e uma referência para uma função capaz de encontrar os conflitos e devolver o conjunto de contribuintes (Algoritmo 13, apresentado a seguir). As observações não são precisam ser passadas para esse algoritmo porque elas estão disponíveis na variável global OBS.

- No passo **P4**, Linha 11, do Algoritmo 2, ao invés de chamada a função TP com os parâmetros: $SD, COMP - (H(n) \cup \{s\}), OBS$, são passados somente a variável representando o modelo do programa, M, e o conjunto $H(n) \cup \{s\}$.

A função TP passada como parâmetro para o algoritmo *Minimal Hitting Set* gera os conjuntos de contribuintes através dos algoritmos de propagação apresentados na Seção 7.1.4. O Algoritmo 13 descreve como funciona a função TP que implementamos. Quando essa função é chamada pelo algoritmo *Minimal Hitting Set*, são passados os componentes que serão suspensos (componentes nos quais não é feita a propagação). Após suspender esses componentes, os valores das observações (variável global OBS) são aplicadas às conexões e a propagação de valores é executada. Caso algum conflito de valores seja detectado durante a propagação, o conjunto de contribuintes é devolvido para o algoritmo *Minimal Hitting Set* utilizar para rotular um nó da árvore. Esse algoritmo usa o Algoritmo 14, que é responsável por atribuir as observações atuais (mantidas na variável global OBS) às conexões do sistemas.

O Algoritmo 15 encontra as hipóteses de falha utilizando um modelo do programa M e as observações atuais. Esse algoritmo simplesmente executa o algoritmo *Minimal Hitting Set* considerando as modificações citadas no início dessa seção, que devolve as hipóteses de falha.

Algoritmo 13: $TP(M, suspendedComps)$

Entrada: M: modelo do programa com componentes e conexões,
$suspendedComps$: conjunto de componentes que serão suspensos,
$prevSuspComps$: uma variável global, indicando os componentes que foram suspensos na chamada anterior à função TP,
OBS: variável global que contém as últimas observações usadas pelo depurador.
Saída: conjunto de contribuintes ou $\emptyset$.

```
▷ tira os componentes anteriores do estado de suspensão.
1  para cada cinprevSuspComps faça
2      remover a suspensão do componente c ;
3  fim
▷ faz a suspensão dos componentes passados como parâmetros.
4  prevSuspComps ← suspendedComps ;
5  para cada c ∈ suspendedComponents faça
6      suspender o componente c ;
7  fim
8  aplicaObservacoes(M, OBS) ;
9  contribuintes ← propagarPendentes() ;
10 devolva contribuintes ;
```

Algoritmo 14: aplicaObservacoes(M, OBS)

Entrada: M: modelo do programa com componentes e conexões,
OBS: variável global que contém as últimas observações usadas pelo depurador.

```
1  para cada v ∈ OBS faça
2      n ← propriedade "nome" da variável v ;
3      v ← propriedade "valor" da variável v ;
4      conexao ← busca uma conexão em M com o nome n ;
5      atribui o valor v à conexão conexao ;
6  fim
```

7.1.6 Discriminação de hipóteses

Após serem encontradas as hipóteses de falha utilizando um modelo M, que pode conter componentes abstratos, é possível fazer a discriminação de hipóteses. Além de informar os valores para variáveis o aluno também pode optar por fazer o refinamento de componentes abstratos. A seguir apresentamos como essas operações foram implementadas.

Algoritmo 15: HPD-Diagnosis(M)

Entrada: M: modelo baseado em valor com diversos níveis de abstração,
OBS: variável global que contém as últimas observações usadas pelo depurador.
Saída: Um conjunto de hipóteses de falha.

```
1  hyp ← MinimalHittingSet-MBD(M, TP) ;
```

Refinamento de componente abstrato

Para facilitar a implementação da funcionalidade utilizada para refinar de componente abstrato utilizamos uma abordagem que não requer alteração na estrutura do modelos, mas somente a alteração no valor de uma propriedade que todos componentes abstratos possuem, informando se eles foram refinados ou não. Dessa forma, podemos controlar durante a geração dos conjuntos de contribuintes quais os componentes que serão considerados como falhos, isto é, o componente abstrato ou os componentes internos. O Algoritmo 16 exemplifica como é esse processo de refinamento.

Algoritmo 16: refinar(CA)

Entrada: CA: um componente abstrato.
Saída: O componente CA refinado.
1 atribuir $verdadeiro$ à propriedade $refined$ do componente abstrato CA ;

Num modelo sem componentes abstratos, para cada valor atribuído a uma conexão durante a propagação é armazenada uma referência para o componente que gerou aquele valor. Quando ocorre um conflito de valores, todos os componentes utilizados para gerar os valores conflitantes são usados para compor o conjunto de contribuintes. Esse comportamento muda quando a conexão na qual foi detectado o conflito está conectada a algum componente interno de um componente abstrato. Suponha que C seja um componente interno de um componente abstrato CA não refinado. Ao detectar uma falha envolvendo o componente C, a conexão manterá a informação de que a falha está em CA e não em C. Dessa forma, os conjuntos de contribuintes (e consequentemente as hipóteses de falha) sempre farão referência ao componente CA, enquanto ele não for refinado.

Observe que no nosso modelo é possível que existam componentes dentro de componentes sem restrições de profundidade. Visto que, a estrutura do modelo não é realmente alterada quando é feito o refinamento de um componente abstrato, devemos tomar o cuidado ao obter o componente que fará parte do conjunto de contribuintes quando for detectado um conflito, de ao invés de procurar pelo componente abstrato no qual C é um componente interno, devemos obter uma lista formada por $\langle C_1, C_2, \cdots C_n \rangle$, tal que C é um componente interno de C_1 e C_i é um componente interno de C_{i+1}, $1 \leq i < n$, e procurar dentre os componentes dessa lista pelo componente C_i com o maior valor para i e que ainda não foi refinado. Dessa forma, sempre garantimos que as hipóteses de falha serão geradas com referência para os componentes do nível mais alto de abstração que não foram refinados. O Algoritmo 17 mostra como é feito esse processo. A propriedade *parent* de um componente C qualquer mantém a referência para um componente abstrato CA no qual C é um componente interno, ou nulo caso tal componente CA não exista.

O Exemplo 7.1, mostra com é usado o Algoritmo 17.

Exemplo 7.1 - Exemplo de como obter um componente abstrato não refinado de nível mais alto. Sejam A, B e C três componentes, dos quais A e B são componentes abstratos e não foram refinados, e C é um componente não abstrato. Suponha que B seja um componente interno

Algoritmo 17: componenteAbstratoNaoRefinadoNoNivelMaisAlto(C)

Entrada: C: um componente qualquer do modelo;
Saída: Um componente CA no nível mais alto de abstração, não refinado, no qual C é um componente interno CA, ou C é componente interno de um dos componentes internos de CA, e assim por diante. Caso tal CA não seja encontrado, deverá ser devolvido o valor *nulo*.

▷ *abs* conterá todos os componentes abstratos da árvore de abstração que contém C como componente interno ou que contém algum componente interno que contém C como componente interno, e assim por diante.

1 $abs \leftarrow \emptyset$;
2 $CA \leftarrow parent$ de C;
▷ no final desse laço, o primeiro elemento de *abs* é o componente abstrato no nível de abstração mais alto da árvore de abstrações.
3 **enquanto** *CA for diferente de nulo* **faça**
4   adicionar no começo da lista *abs* o componente CA ;
5     $CA \leftarrow parent$ de CA ;
6 **fim**
▷ os elementos de *abs* devem ser acessados a partir da primeira posição.
7 **para cada** $CA \in abs$ **faça**
8   **se** *CA não refinado* **então**
9   **devolva** CA ;
10 **fim**
11 **devolva** nulo ;

de A e C seja um componente interno B. Ao executar o Algoritmo 17 passando como parâmetro o componente C será devolvido o componente A. Suponha agora que o componente A seja refinado. Ao executar novamente o Algoritmo 17 passando como parâmetro o componente C será devolvido dessa vez o componente B, que agora é o que está no nível de abstração mais alto e não foi refinado. □

Informar valores para variáveis

A operação de informar valores para variáveis consiste basicamente em adicionar ao conjunto corrente de observações uma nova observação e obter as hipóteses de falha novamente. Visto que os valores informados pelo aluno podem ser excluídos, mas não as entradas e saídas do caso de teste usado na depuração, é importante diferenciar esses dois conjuntos de informações usados para compor as observações. Substituindo as chamadas para o Algoritmo 10 pela chamada para o Algoritmo 18 é possível obter o comportamento desejado. No Algoritmo 18 é utilizada uma variável global, *obsDoAluno*, que mantém as observações fornecidas pelo aluno e que podem ser alteradas a qualquer momento. Uma vez que esse conjunto de informações foi alterado, bastar executar novamente o Algoritmo 15, para gerar as hipóteses de falha considerando esses novos valores.

Na seção a seguir, apresentamos o Dr. Java Pro: um ambiente para o desenvolvimento de programas que disponibiliza o nosso sistema HPD para fazer a depuração automática de programas.

Algoritmo 18: transformarEntradasESaidasEmObservacoes(TC)

Entrada: TC: o caso de teste a partir do qual serão geradas as observações,
obsDoAluno: variável global que contém as observações informadas pelo aluno.

1 $entradas \leftarrow$ entradas do TC ;
2 $saidas \leftarrow$ saídas do TC ;
3 $OBS \leftarrow \emptyset$;
4 $OBS = OBS \cup entradas$;
5 $OBS = OBS \cup saidas$;
6 $OBS = OBS \cup obsDoAluno$;
7 **devolva** OBS ;

7.2 Dr. Java Pro: uma extensão do Dr. Java com depurador automático

O Dr. Java (http://www.drjava.org/) é um ambiente utilizado para a construção de programas, que tem sido adotado por diversos professores do Departamento de Ciência da Computação do IME/USP na disciplina de Introdução à Programação, nos últimos semestres. Basicamente, no Dr. Java os alunos podem realizar as principais tarefas correlatas à construção de programas, tais como: editar, compilar, executar e depurar seus programas. Devido à facilidade de uso desse ambiente e também pela adoção em disciplinas introdutórias de programação, optamos por estendê-lo com o depurador automático de programas proposto nesse trabalho, o sistema HPD. Chamamos essa extensão do Dr. Java com o sistema HPD de Dr. Java Pro. Além da integração com o depurador automático, foram feitas algumas modificações para deixar o ambiente mais simples e intuitivo de ser utilizado por aprendizes de programação, a saber:

- Remoção da interface do ambiente dos botões de atalho para funcionalidades pouco utilizadas por aprendizes de programação. As funcionalidades não foram removidas e podem ser acessadas através dos menus do ambiente.

- Disponibilização de uma coletânea de problemas previamente definidos e acessíveis por um menu específico de atividades. Ao iniciar o Dr. Java Pro, é feito o acesso a um repositório de problemas que disponibilizamos na Internet, que são instalados automaticamente no computador do aluno, onde o ambiente está sendo executado. Uma vez que os problemas estão disponíveis na máquina do aluno, ele pode abri-los para construir os seus programas.

- Padronização da forma como os problemas são definidos e apresentados aos alunos, contendo os seguintes componentes: enunciado, exemplos, casos de teste e programas de exemplo.

- Definição de um forma de execução e apresentação dos casos de teste para os problemas pré-definidos. Apesar do Dr. Java permitir que os casos de teste sejam construídos e executados, a forma como isso é feito é trabalhosa e pouco intuitiva para aprendizes de programação.

- Tradução do inglês para o português de todos os menus do ambiente.

A seguir, mostramos um exemplo de depuração de programas utilizando o Dr. Java Pro, e também aproveitamos para descrever as principais funcionalidades e telas desse ambiente. Utilizaremos o Programa 1 como exemplo, que foi construído por um aluno para resolver o problema da *Ordem Crescente* (Exemplo 1.1 apresentado no Capítulo 1).

Após abrir um problema no Dr. Java Pro, todas as informações referentes a esse problema são apresentadas em regiões do ambiente e o aluno pode começar a trabalhar na sua solução. A Figura 7.2, mostra a tela principal do Dr. Java Pro com o problema da *Ordem Crescente* já carregado. As seguintes regiões são apresentadas nessa figura:

- **Região A**, que mostra os arquivos de código fonte da problema atual;
- **Região B**, que é o local onde o aluno constrói e edita seu programa. Observe que o Programa 1 é apresentado nessa região;
- **Região C**, onde é apresentado o enunciado do problema;
- **Região D**, na qual são apresentadas as abas de ferramentas e, nesse caso, está visível uma nova aba disponibilizada no Dr. Java Pro que mostra os casos de teste para o problema da *Ordem crescente*. Também vale citar que os casos de teste somente poderão ser executados caso o programa do aluno seja compilado sem erros.

Na aba de casos de teste é apresentada uma tabela que mostra as informações sobre os casos de teste do problema, com as seguintes colunas:

- **#CT,** que define um identificador para cada caso de teste;
- **Entradas**, que são as entradas especificadas para cada caso de teste;
- **Saídas esperadas**, que são as saídas esperadas que foram definidas para cada caso de teste;
- **Saídas obtidas**, que são as saídas obtidas com a execução do programa com as entradas especificadas em cada caso de teste;
- **Tempo de execução**, que é o tempo consumido (em milissegundos) para executar o programa com as entradas de cada caso de teste;
- **Status**, que define o status da execução do programa para cada caso de teste TC, representado por cada linha da tabela, e que podem ter um dos seguintes valores: *Não executado*, quando o programa ainda não foi testado com o caso de teste TC; *Passou*, quando o programa foi executado com sucesso para o caso de teste TC e; *Falhou*, quando o programa falhou para o caso de teste TC. Além disso, quando o programa é executado com sucesso para um determinado caso de teste, a linha na tabela que representa esse caso de teste é mostrada na

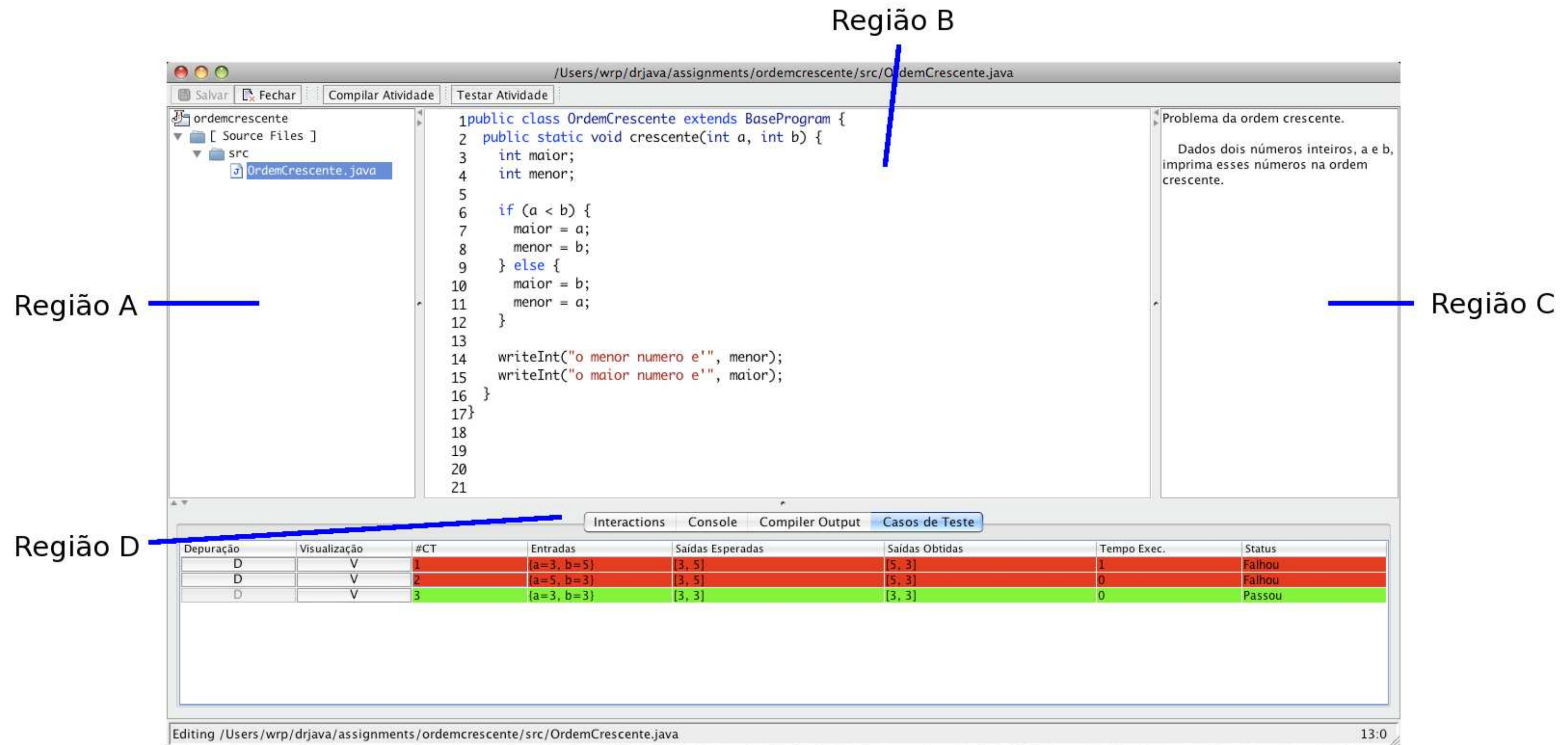


Figura 7.2: Tela principal do Dr. Java Pro, mostrando as informações para o problema da *Ordem Crescente*, apresentado no Exemplo 1.1.

cor verde, enquanto que, para o caso de teste no qual o problema falhou, a linha é mostrada na cor vermelha.

No lado esquerdo de cada linha representando um caso de teste existe um botão marcado com a letra "V", que apresenta os detalhes do caso de teste da linha selecionada, e outro botão marcado com a letra "D", que tem como função executar o depurador automático utilizando as entradas e saídas especificadas pelo caso de teste selecionado. Esse botão fica desabilitado para as linhas dos casos de teste que executaram com sucesso. Note que, pela Figura 7.2, somente um dos três casos de teste disponíveis foi executado com sucesso (TC3).

Figura 7.3: Tela do Dr. Java Pro que mostra as hipóteses de falha no programa do aluno, construído para solucionar o problema da *Ordem Crescente*, e usando o caso de teste TC1.

Vamos utilizar o primeiro caso de teste, que chamaremos de TC1, para encontrar as falhas no programa do editor. No TC1 são definidas as seguintes entradas: $a = 3$ e $b = 5$; e as saídas esperadas são: 3 e 5. Ao clicar no botão "D" da linha da tabela que representa o TC1, o depurador automático usa as informações desse caso de teste selecionado para gerar as seguintes hipóteses de falha:

$$\{\{SelecaoAlternativa\}, \{15, 14\}\}$$

A hipótese $\{SelecaoAlternativa\}$ indica que há uma ou mais falhas no padrão elementar de seleção alternativa, que nesse caso foi usado entre as linhas 6 e 12 do programa; e a hipótese de falha $\{15, 14\}$ indica que as linhas 14 e 15 contém falhas, sendo necessário corrigir essas duas linhas para que o programa seja executado com sucesso para o caso de teste utilizado.

Essas hipóteses de falha geradas são apresentadas em uma nova aba na tela do Dr. Java Pro. A Figura 7.3 mostra essa nova aba, que é dividida nas seguintes regiões:

- **Região D.1**, onde são apresentadas as informações do caso de teste utilizado para fazer a

depuração automática do programa e a saída devolvida pelo programa após ser executado com as entradas do caso de teste em questão;

- **Região D.2**, utilizada para mostrar as hipóteses de falha geradas pelo depurador. Quando uma hipótese de falha é selecionada, as linhas do programas relacionadas a essa hipótese são apresentadas na cor amarela. Note que podem aparecer nas hipóteses de falha tanto linhas isoladas do programa quanto componentes abstratos (que compreendem um conjunto de linhas do programa);
- **Região D.3**, que apresenta uma tabela utilizada para mostrar os valores informados para variáveis durante a discriminação de hipóteses;
- **Região D.4**, onde são apresentados os seguintes botões:
  - **Editar**, que permite ao aluno interromper o processo de depuração e voltar a edição do programa.
  - **Informar valores para variáveis**, que permite ao aluno informar valores para as variáveis relacionadas à hipótese de falha selecionada.
  - **Refinar componente abstrato**, que permite ao aluno refinar um ou mais componentes abstratos pertencentes à hipótese de falha selecionada.

A partir de um conjunto de hipóteses de falha gerado pelo depurador automático, o aluno tem a possibilidade de informar valores para variáveis ou refinar componentes abstratos relacionados a uma hipótese selecionada. Após executar uma dessas ações, o depurador poderá gerar novas hipóteses de falha, a partir das quais o aluno poderá novamente executar as ações citadas. Esse processo poderá ser feito até que seja gerada somente uma hipótese de falha que o aluno poderá utilizar para corrigir o programa. A Figura 7.4 mostra um grafo que relaciona ações do aluno com as hipóteses de falha geradas pelo depurador automático, durante a depuração do Programa 1 com o caso de teste TC1. Nessa figura, os nós são rotulados pelas hipóteses de falha devolvidas pelo depurador automático, identificados por letras de A até E, e os arcos denotam as ações executadas com uma hipótese de falha selecionada, gerando um novo conjunto de hipóteses de falha (nós em que os arcos chegam). Note que não mostramos nesse grafo todas as possibilidades. Por exemplo, o aluno poderia optar por fazer o refinamento do componente abstrato de *SelecaoAlternativa* logo na primeira ação, fazendo com que as seguintes hipóteses de falha (citadas no Capítulo 1) fossem geradas:

$$\{\{6\},\{7,8\},\{14,15\},\{8,15\},\{7,14\}\}$$

mas para manter esse exemplo simples, optamos por informar valores para variáveis logo no primeiro passo, reduzindo imediatamente o conjunto de hipóteses de falha, como mostramos a seguir. A

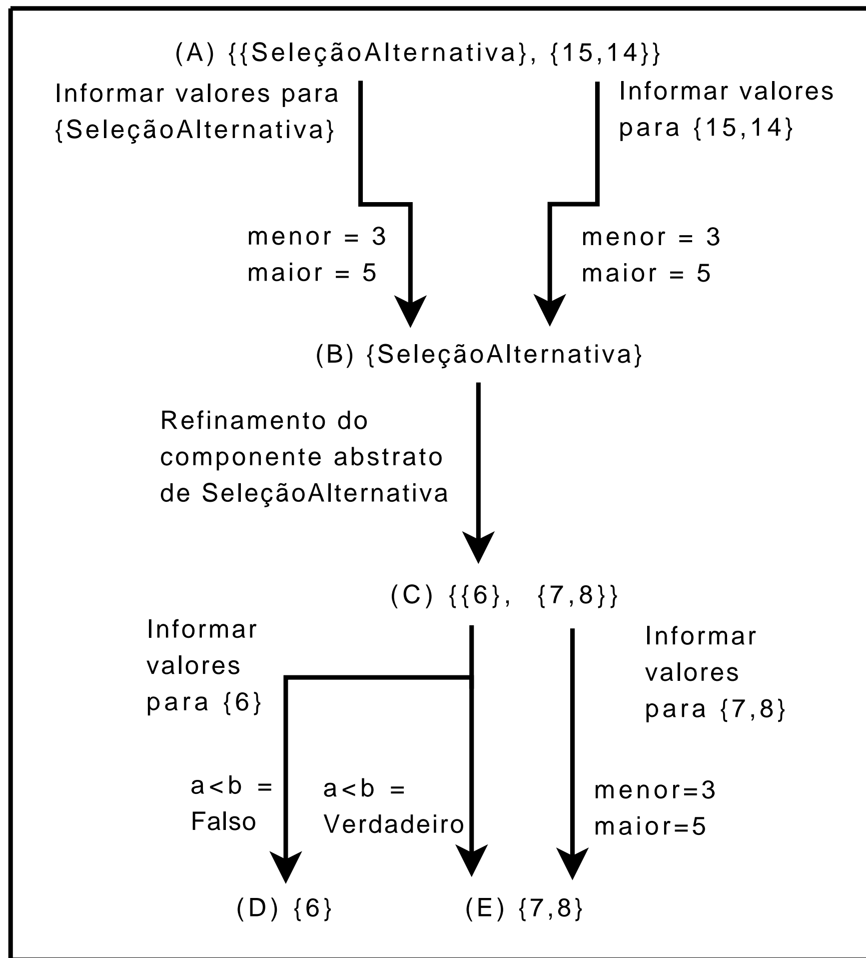
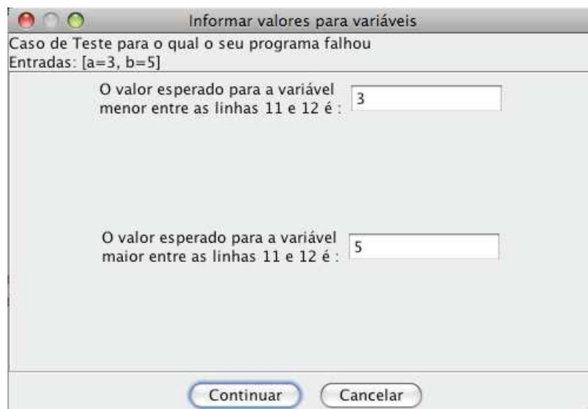


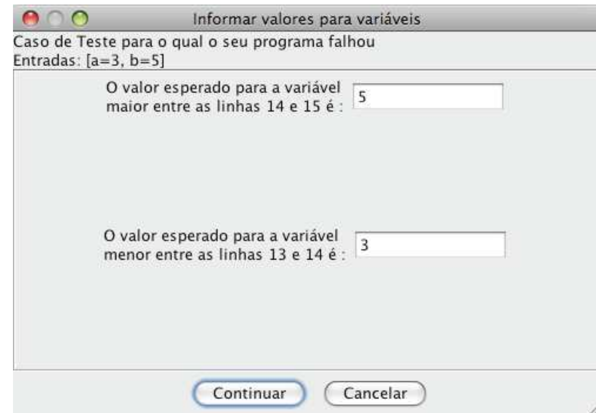
Figura 7.4: Alguns dos possíveis caminhos que um aluno pode seguir durante a depuração do Programa 1, com o caso de teste TC1.

partir do conjunto de hipóteses (A), é possível de selecionar uma das duas hipóteses de falha para informar valores para variáveis. Independente da hipótese de falha escolhida, é possível informar valores para as variáveis *menor* e *maior*, porém, em diferentes linhas do programa. Para garantir que serão informados valores esperados corretamente, vamos supor que o aluno usou a variável *menor* para armazenar o menor valor recebido na entrada e a variável *maior* para armazenar o maior valor recebido na entrada. Dessa forma, para o caso de teste selecionado devemos ter $menor = 3$ e $maior = 5$. A Figura 7.5 mostra as telas utilizadas para informar valores para as variáveis relacionadas aos conjuntos de hipóteses de falha: $\{\{SelecaoAlternativa\}$ (Figura 7.5a) e $\{15, 14\}\}$ (Figura 7.5b). Na tela utilizada para informar valores para variáveis são apresentadas todas as variáveis relacionadas à hipótese de falha selecionada e as linhas nas quais espera-se que essas variáveis possuam os valores informados. Para as variáveis que recebem valores inteiros é apresentado um campo texto onde deve ser informado o valor esperado para cada variável e, para as variáveis lógicas é apresentado um campo que permite selecionar as opções *Verdadeiro* ou *Falso*.

Na parte superior da tela são apresentadas as informações do caso de teste utilizado na depuração.



(a) Tela utilizada para informar valor para as variáveis relacionada à hipóteses de falha $\{SelecaoAlternativa\}$.



(b) Tela utilizada para informar valor para as variáveis relacionada à hipóteses de falha $\{15, 14\}$.

Figura 7.5: Telas utilizadas para informar valores para as variáveis relacionadas ao conjunto de hipóteses de falha: $\{\{SelecaoAlternativa\}, \{15, 14\}\}$.

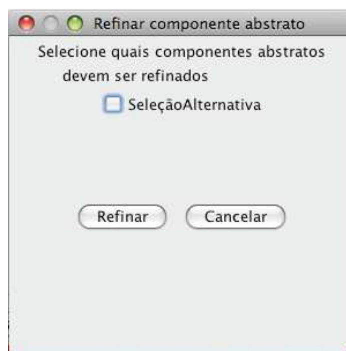


Figura 7.6: Tela utilizada para refinar o componentes abstrato de $SelecaoAlternativa$.

Independente da hipótese de falha utilizada para informar valores para variáveis, o depurador automático gera somente uma hipótese de falha, contendo o componente abstrato $SelecaoAlternativa$ (nó (B) na Figura 7.4). Como essa hipótese de falha contém somente um componente abstrato, vamos verificar as falhas nas linhas que compõem esse componente, executando a operação de refinamento. A Figura 7.6 mostra a tela utilizada para fazer o refinamento de componentes abstratos, considerando a hipótese de falha selecionada, que nesse caso contém somente o componente abstrato de $SelecaoAlternativa$. Se essa hipótese possuísse mais componentes abstratos, todos eles seriam apresentados e poderíamos escolher um ou mais componentes para refinar.

Ao refinar o componente abstrato $SelecaoAlternativa$, o depurador automático gera novamente as hipóteses de falha considerando os componentes internos desse componente abstrato. As novas hipóteses de falha geradas (nó (C) da Figura 7.4) são:

$$\{\{6\},\{7,8\}\}$$

A hipótese de falha $\{6\}$ indica que a condição do `if` na linha 6 deve estar falha, enquanto que, a hipótese $\{7,8\}$ indica que as linhas 7 e 8 estão com falhas, sendo que, para prover uma solução para o programa considerando esse última hipótese de falha é necessário corrigir essas duas linhas. A Figura 7.7 mostra como essas hipóteses de falha são apresentadas pelo Dr. Java Pro.

Figura 7.7: Hipóteses de falha geradas após refinar o componente abstrato de Seleção Alternativa.

Nesse momento, é possível somente executar a operação de informar valores para variáveis, considerando as hipóteses geradas pelo depurador automático. Porém, é necessário pensar em um caminho lógico para tentar encontrar a falha no programa e selecionar a hipótese de falha desejada conforme esse caminho escolhido. É possível escolher os dois seguintes caminhos:

1. *Resolver a falha na condição da seleção da Linha* 6. Nesse caso, devemos informar se o valor esperado na avaliação da condição da linha 6 é *Falso* ou *Verdadeiro*. Se informarmos que o valor esperado para a condição é *Falso*, o depurador irá manter somente a hipótese de falha $\{6\}$, pois, nesse caso, o programa não está fazendo o que foi esperado (para o caso de teste no qual $a = 3$ e $b = 5$, a avaliação da condição $a < b$ é verdadeira, contradizendo o que foi informado e indicando que a linha 6 está com falha). Dessa forma, essa linha deve ser usada para corrigir a falha no programa. Caso informarmos que o valor esperado para essa condição é *Verdadeiro*, o depurador eliminará essa hipótese de falha, pois, o programa está fazendo o que é esperado.

2. *Resolver as falhas nas atribuições das Linhas* 7 *e* 8. Nesse caso, devemos informar os valores esperados para as variáveis *maior* (entre as linhas 7 e 8) e para a variável *menor* (entre as linhas 8 e 9), que devem possuir os valores: $menor = 3$ e $maior = 5$.

Em uma sessão de depuração com um aluno, nessa situação, ele deve decidir qual caminho seguir e para que possam ser informados valores esperados para as variáveis corretamente, o aluno tem que pensar nas suas intenções ao construir o programa. Por exemplo, no caso desse problema em específico, poderia ser encontrada uma solução usando na condição da linha 6 a expressão $a < b$ ou a

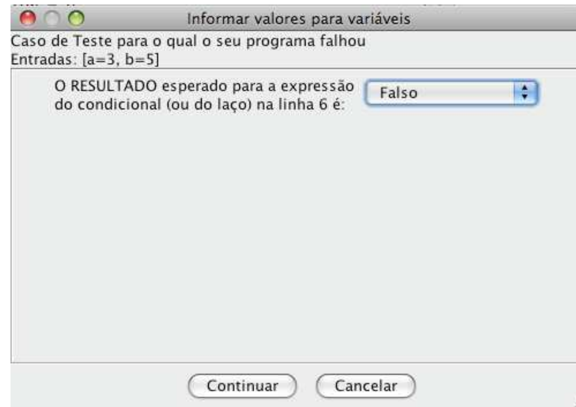


Figura 7.8: Tela utilizada para informar o valor esperado para a avaliação da condição $a < b$ na linha 6 do Programa 1.

expressão $a > b$. Vamos supor que a intenção de um aluno seja fazer $a < b$ e, dado os valores: $a = 5$ e $b = 3$, informados no caso de teste TC1, então o resultado esperado da avaliação dessa expressão é *Verdadeiro*. Ao informar esse valor, o depurador entende que essa linha está funcionando de acordo com a intenção do aluno e essa hipótese de falha é removida, restando somente a hipótese de falha $\{7, 8\}$. Por outro lado, caso a intenção do aluno era fazer $a > b$, e na tela de informar valores para variáveis for informado o valor *Falso* como o resultado da avaliação $a < b$, o depurador entende que a falha deve estar na Linha 6, eliminando a hipótese de falha $\{7, 8\}$. Assim, é necessário que o aluno sempre pense em suas intenções para escolher qual hipótese de falha discriminar e também para informar valores para variáveis.

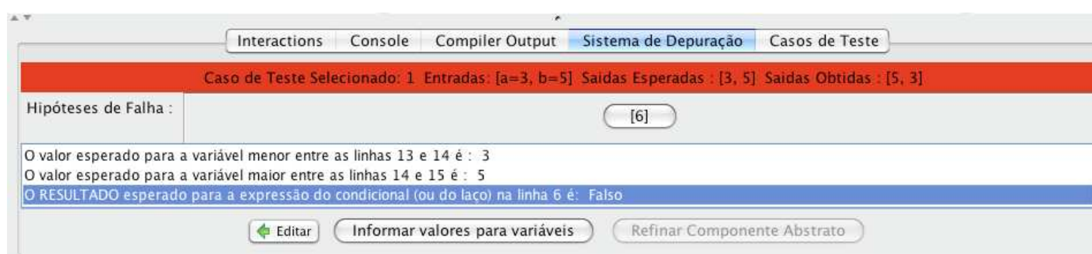


Figura 7.9: Tela mostrando os valores informados para variáveis durante a sessão de depuração.

Para finalizar o nosso exemplo, vamos selecionar a hipótese de falha $\{6\}$ e informar que o valor esperado para a condição é *Falso*. A Figura 7.9, mostra o resultado final da sessão de depuração, apresentando somente a hipótese de falha $\{6\}$. Nesse momento, espera-se que um aluno que tenha chego até esse ponto seja capaz de reconhecer a falha na condição da linha 6 e que consiga modificar o programa, substituindo o operador “$<$” pelo operador “$>$”, corrigindo assim a falha no programa.

7.3 Considerações finais

Nesse capítulo, apresentamos como foi feita implementação do depurador automático de programas, o sistema HPD apresentado no Capítulo 6. Também foram apresentados diversos detalhes a respeito da implementação do gerador de modelos e do mecanismo de propagação. Além disso, mostramos como o nosso depurador automático foi usado para estender o Dr. Java, um ambiente utilizado para a construção de programas, gerando um novo ambiente chamado de Dr. Java Pro. Também apresentamos uma sessão de depuração de programas utilizando essa nova ferramenta.

No próximo capítulo, apresentaremos como foi feita a verificação da nossa implementação e os resultados da avaliação da nossa ferramenta, feita através de experimentos com alunos de uma disciplina de Introdução à Programação.

Parte III

Avaliação Experimental e Conclusões

Capítulo 8

Verificação e avaliação da Ferramenta Dr. Java Pro

Neste capítulo, verificamos a implementação do depurador automático de programas, isto é, mostramos que a implementação do algoritmo especificado no Capítulo 6 está correta e devolve as hipóteses de falha esperadas. Apresentamos os resultados de uma avaliação didática preliminar da ferramenta proposta, realizada com alunos de uma disciplina de Introdução à Programação que teve como objetivo avaliar o uso do depurador automático Dr. Java Pro por um grupo selecionado de alunos. Além disso, mostramos os resultados de uma avaliação do Dr. Java Pro realizada pelos alunos através de um questionário.

No contexto desse capítulo, o termo *verificação* é empregado no uso de testes que garantem a corretude da implementação do Algoritmo 7; o termo *avaliação* é empregado nas análises das interações de alunos com o Dr. Java Pro.

8.1 Verificando a implementação do Dr. Java Pro

Durante o desenvolvimento, para verificar a corretude da implementação da ferramenta de depuração automática, foi especificado um conjunto de "problemas de teste", compostos por problemas de programação e suas respectivas soluções cujas falhas devem ser identificadas pelo sistema de depuração[1]. Como no trabalho de Delgado (2005), analisamos o conjunto de hipóteses gerado pelo depurador automático e as comparamos com as saídas dos problemas de teste.

A escolha do conjunto de problemas de teste usados na verificação do Dr. Java Pro (incluindo as soluções com falhas) foram extraídas de soluções reais de alunos em provas presenciais aplicadas em turmas de anos anteriores da disciplina de Introdução à Programação ministradas pelos docentes do Departamento de Ciência da Computação do IME/USP.

Para executar os problemas de teste, utilizamos um arcabouço chamado JUnit (`http://www.junit.org/`), que é muito utilizado no desenvolvimento de sistemas dirigido por testes [4]. Ba-

[1] Chamamos de "casos de teste", os exemplos de entradas e saídas para o programa do aluno; "problemas de teste", os exemplos de entradas e saídas para o sistema Dr. Java Pro, sendo que, nesse caso, as entradas são: um programa, um caso de teste para o qual o programa falhou, e a saída é o conjunto de hipóteses de falha do programa; e de "problemas de programação" os problemas dados para os alunos resolverem.

sicamente, o JUnit executa os casos de teste e, se a saída obtida da execução de cada caso de teste for igual ao resultado esperado, o JUnit mostra na tela uma barra na cor verde informando que o teste obteve sucesso. Caso contrário, ele mostra uma barra vermelha indicando que o caso de teste falhou. É interessante notar que essa mesma ideia de testes de programas do JUnit foi implementada no Dr. Java Pro para verificar os programas dos alunos (Seção 7.2). Para verificar a corretude do depurador, usamos o JUnit de uma forma diferente: a saída esperada deve pertencer (e não ser igual) à saída gerada pela execução do depurador automático. Isso porque, conforme visto no Capítulo 6, o sistema de depuração automática devolve um conjunto de **todas** as hipóteses de falha possíveis, sendo que algumas delas são difíceis de serem previstas *a priori* pelos problemas de teste. Assim, um problema de teste é executado com sucesso se a falha introduzida no programa solução pertencer ao conjunto de hipóteses de falha geradas pelo depurador. Para facilitar a verificação, implementamos um modelo de arquivo para padronizar a configuração das entradas e saídas utilizadas na verificação do depurador (no Apêndice C detalhamos esse modelo de arquivo).

O uso de problemas de teste também foi fundamental para fazer a verificação da nossa implementação durante o desenvolvimento do depurador. Através dessa abordagem, foi possível garantir que ao implementarmos uma nova funcionalidade, aquelas que já estavam implementadas continuariam funcionando. Isso foi feito selecionando um problema de teste para cada situação de depuração que implementamos. Por exemplo, o problema Max (Apêndice B) foi usada para verificar se o sistema de depuração automática é capaz de identificar falhas na instrução seleção alternativa (*if-then-else*). Assim, já durante o desenvolvimento, garantimos o sucesso de todos os problemas de teste utilizados, ou seja, **todos os problemas de teste passaram pelo JUnit com sucesso**. A Tabela 8.1, contém os problemas de teste que foram utilizados para verificar a nossa implementação e uma descrição das falhas introduzidas nas soluções.

Problema	**Falhas**
Maratona	uso de constante ou operadores incorretos em expressão (F3, F4, F5)
Max	uso incorreto de operadores na condição do comando de seleção (F12); inversão de ações entre os ramos *then* e *else* (F13)
CalculaPremio	uso incorreto de operadores na condição de seleções (F12); inversão de ações entre os ramos *then* e *else* (F13); expressão errada no corpo de seleção (F14)
VerificaPar	uso incorreto do operador de resto da divisão (F3)
Mediana	uso incorreto de operadores na condição de seleções aninhadas (F14)
Multa	atribuição incorreta do valor de variável (F1, F2)
SomaDigitos	uso de operador incorreto na condição de laço (F12)
Triangulo	erro em expressões contendo operadores lógicos “e” e “ou” (F6, F7)
Palindromo	atribuição incorreta do valor de variável no corpo do comando `while` (F14)

SomaNNumerosMain	expressão incorreta para manipular valores lidos no corpo de comando `while` (F14)
SomaNNumerosComForMain	variação do "SomaNNumerosMain" utilizando um laço do tipo *for*
Calculadora	erro na passagem de parâmetros na chamada de métodos (F16, F17); expressão incorreta utilizada em chamada de método (F16)
SomaMultiplos	uso incorreto de variável de controle em expressões no corpo de laço do tipo `for` (F14)
Perfeito	uso incorreto da variável de controle de laço do tipo `for` no corpo de uma seleção dentro do laço (F14)
SomaNumeros	para dois laços do tipo `for` encadeados, uso de expressão incorreta no corpo de laço mais interno (F14)
TesteImpl	uso incorreto de operadores de incremento e decremento (F9)
MaxDigitosEmSeq	instrução *return* com expressão incorreta no corpo de seleção (F4) ; expressão incorreta no corpo de seleções encadeadas, dentro de um laço (F4, F14)
Maximiza	para dois laços do tipo *while* encadeados, uso de expressão incorreta no corpo de laço mais interno (F14, F4)
NumeroSegmentos	uso incorreto de variável de leitura no corpo de laço do tipo *for* (F14, F4)

Tabela 8.1: Lista de problemas de teste e falhas introduzidas para fazer a verificação da implementação do sistema de depuração automática. Cada falha contém uma referência (Fi) para as falhas listadas na Tabela 2.2.

8.2 Avaliação do uso do Dr. Java Pro por um grupo de alunos

Para avaliar o uso da ferramenta Dr. Java Pro, foi selecionado um grupo de alunos da disciplina de Introdução à Programação (oferecida ao curso de licenciatura em Física da USP) e dois problemas de programação envolvendo aspectos fundamentais para o depurador baseado em modelos para serem resolvidos em 2 aulas práticas de avaliação. Em geral, problemas de programação para serem resolvidos pelo Dr. Java Pro podem ser de dois tipos:

1. **Resolva**, nos quais os alunos devem construir uma solução a partir de um esqueleto de programa vazio. Nesse caso, os alunos recebem um esqueleto de programa que deve ser completado de forma a solucionar o problema descrito no enunciado. Na maioria dos casos, o esqueleto do programa já declara algumas variáveis necessárias para resolver o problema e comentários que ajudam o aluno a estruturar sua solução. Apesar dessas sugestões, os alunos têm liberdade para incluir, excluir ou modificar as variáveis do problema.

2. **Modifique**, nos quais é dado ao aluno um programa pronto, contendo uma ou mais falhas. Para esse tipo de problema, os alunos devem detectar e corrigir falhas num programa que aparentemente resolve o problema.

Cada problema de programação é descrito por um conjunto de atributos, a saber:

- **Nome**, especifica o nome do problema de forma mnemônica.
- **Enunciado**, descreve o problema a ser resolvido e uma sugestão de estratégia.
- **Tipo**, identifica se o aluno deve desenvolver o programa (tipo = **Resolva**) ou se ele deve modificar um programa já existente (tipo = **Modifique**).
- **Conjunto de casos de teste**, indicam as entradas e saídas do programa, que o aluno pode utilizar para auxiliá-lo a resolver o problema e que será utilizado pelo sistema de depuração automática.
- **Uma classe pré-definida**, indica o nome da classe e do método que contém o esqueleto do programa no qual o aluno deverá desenvolver seu programa. Essa informação é importante para fazer a execução dos casos de teste e também para o depurador automático.
- **Justificativa**, que descreve o objetivo didático do problema para o professor.

Durante a avaliação, os alunos utilizaram computadores do tipo PC, com sistema operacional Debian Linux e uma versão pré-instalada e configurada da ferramenta Dr. Java Pro. Enquanto os alunos manipulavam a ferramenta, eram gerados *logs* para certos tipos de eventos executados na ferramenta. A lista desses eventos que geram *log* é apresentada na Tabela 8.2. Ao final de cada aula o arquivo de *log* em cada computador era recolhido e armazenado. A partir desses *logs* foram feitas as análises descritas nessa seção.

A seguir, descrevemos os problemas de programação utilizados na avaliação preliminar realizada e também uma análise do resultados obtidos.

Problema P1: Cálculo do número par

Esse problema é do tipo **Resolva**, ou seja, é dado ao aluno um esqueleto de programa vazio, que ele deve completar de forma a solucionar o problema. A Tabela 8.3 descreve os atributos do Problema P1, conforme o padrão adotado. O esqueleto fornecido para o Problema P1 é mostrado no Programa 4 que contém o enunciado comentado e, dentro do método, comentários para explicar o uso das variáveis declaradas, nesse caso, uma recomendação para o aluno usar apenas uma vez a instrução `return`.

Identificador	**Descrição do Evento**
ST0	Projeto carregado.
ST1	Código fonte carregado (registra o código fonte).
ST2	Compilação finalizada com sucesso.
ST3	Compilação finalidade com erros (registra os erros).
ST4	Caso de teste executado (registra o resultado da execução).
ST5	Inicializada a depuração (registra o caso de teste usado).
ST6	Usuário selecionou uma hipótese de falha (registra a hipótese selecionada).
ST7	Usuário optou por informar valores para variáveis (registra a hipótese de falha utilizada).
ST8	Usuário informou valor para variáveis (registra variáveis e valores informados).
ST9	O depurador encontrou falhas no programa (registra as hipóteses encontradas).
ST10	O usuário optou por remover um valor informado para uma variável (registra o valor e a variável).
ST11	O usuário optou por refinar os componentes de uma hipótese (registra a hipótese).
ST12	O usuário optou por refinar um componente abstrato (registra o componente abstrato) .
ST13	O usuário finalizou o processo de depuração.
ST14	O conjunto de casos de teste finalizou a execução (registra o status da execução).

Tabela 8.2: Eventos do Dr. Java Pro que geram *log*. Esses eventos serviram como base para fazer a análises experimentais da ferramenta.

O Problema P1, além de ter como objetivo a justificativa descrita na Tabela 8.3, permite que o aluno reforce os seguintes conceitos de programação:

- o uso do operador resto da divisão;
- o uso de uma variável resultado para armazenar o valor que deverá ser devolvido apenas no final do programa;
- a semântica do comando *if-then-else* e
- a correção de falhas no comando *if-then-else* em que tanto a modificação da condição como a troca dos comandos *then-else* podem corrigir o programa (conforme visto no Exemplo 1.1 do Cap. 1), que chamaremos de *correção dual para if-then-else*.

Foram analisados os logs de 30 alunos, sendo que todos os programas gerados pelos alunos estão sintetizados na Tabela 8.4. Note que nessa tabela, apenas os Programas P1f, P1g, P1h e P1i estão

Nome:	*Cálculo do número par*
Enunciado:	*Dado um número inteiro n, calcule n/2 se n for par ou (n-3)/2, caso contrário*
Tipo:	Resolva
Casos de teste:	**Identificador** / **Entradas** / **Saídas**: CT1 / 10 / 5; CT2 / 1 / -1; CT3 / 0 / 0; CT4 / 23 / 10
Classe:	CalculoPar
Justificativa:	Esse problema tem como objetivo introduzir ao aluno a ferramenta Dr. Java Pro e os conceitos de depuração automática. Nesse primeiro contato, devem ser apresentados: o conceito de casos de teste e sua aplicação; os conceitos de hipóteses de falha; a tarefa de discriminação de hipóteses que envolve informar valores para variáveis e refinamento de componentes abstratos.

Identificador	Entradas	Saídas
CT1	10	5
CT2	1	-1
CT3	0	0
CT4	23	10

Tabela 8.3: Atributos do Problema P1.

corretos, sendo que os programas restantes falham em pelo menos um dos casos de teste. A Tabela 8.5 mostra, para cada aluno, a sequência de programas submetidos e casos de teste utilizados na depuração. A Figura 8.1 mostra o número de alunos que escolheram as mesmas sequências de modificações de programas durante a resolução do problema.

Da Tabela 8.5 observamos que dos 11 **alunos que não usaram o depurador**: 9 alunos acertaram o problema com apenas uma rodada de teste (todos esses alunos apresentaram como solução o programa P1f); 1 aluno acertou o problema sem a ajuda do depurador, porém iniciando com o programa P1d e observando os testes que falharam corrigiu o programa para o P1h; e 1 aluno não conseguiu resolver o problema.

Dos 19 **alunos que usaram o depurador**, 90% resolveram o problema corretamente. Observamos que a maioria usou a hipótese de falha apontada pelo depurador uma vez que eles modificaram as linhas correspondentes, a saber (Figura 8.1):

- 6 alunos que iniciaram com o programa P1c, modificaram o bloco *then* (apontado como hipótese de falha pelo depurador para o caso de teste CT2), devolvendo o programa correto P1g (Note que para as entradas do CT2, o bloco *then* é apontado como uma possível falha).
- 10 alunos iniciaram seus programas com o programa P1a:
    - 2 alunos corrigiram a condição (hipótese de falha apontada pelo depurador para o caso de teste CT1 e CT2, respectivamente) e devolveram o programa correto P1f (Note que tanto a hipótese apontada para o CT1 como para o CT2 apontam falha na condição);

Programa 4 Esqueleto do programa Cálculo do Número Par.

```
1   public class CalculoPar {
2     /*
3     * Recebe como entrada um número inteiro n e devolve o resultado da
4     * operação n/2 caso n seja par ou (n-3)/2, caso contrário.
5     *
6     * Você não deve usar nenhuma instrução ‘‘return’’ exceto aquela que
7     * já está no código abaixo.
8     */
9     public int calcula(int n) {
10      /* a variável ‘‘resultado’’ armazena o cálculo das operações definidas
11      * para n par e n ímpar.
12      * Note que esse programa não pode ser compilado antes que seja atribuído
13      * algum valor para a variável resultado, uma vez que ela é devolvida pelo
14      * método através do comando return.  Por isso, vamos inicializá-la com um
15      * valor qualquer, por exemplo:  zero.
16      */
17      int resultado = 0;
18
19      // insira aqui o seu código
20
21      return resultado;
22    }
23  }
```

- 1 aluno inverteu os blocos do *if-then-else* (hipótese de falha apontada pelo depurador para CT1 através da condição) e devolveu o programa correto P1g;
- 5 alunos corrigiram o bloco *else* (hipótese de falha apontada pelo depurador) indo para o programa P1c e em seguida: 4 alunos corrigiram o bloco *then* (hipótese de falha apontada pelo depurador) indo para o programa correto P1g; e 1 aluno usou o caso de teste CT2 indo para o programa P1f;
- e 2 alunos não conseguiram resolver o problema corretamente.

É interessante observar que esses resultados sugerem que as indicações do depurador ajudaram os alunos na correção de seus programas e que, diante do problema da *correção dual para if-then-else*, os alunos em sua maioria, optaram por corrigir os blocos ao invés da condição. Uma justificativa para isso é que alunos iniciantes de programação sentem dificuldade em modificar expressões lógicas, especialmente aquelas envolvendo o operador resto de divisão. Por outro lado, o fato do depurador apontar o bloco que falhou para o caso de teste selecionado, permite que o aluno, sem avaliar a condição, saiba qual bloco ele deve corrigir para aquele caso de teste.

Problema P2: Cálculo da mediana

O Problema P2 é do tipo **Modifique**, ou seja, é dado aos alunos o Programa 5 proposto como uma possível solução para o problema do cálculo da mediana (descrito na Tabela 8.6) mas que

Programa 5 Programa com falha usado para calcular a Mediana entre três números.

```
1   public class Mediana {
2     /**
3     * O método a seguir é utilizado para encontrar a mediana de um conjunto de
4     * 3 números inteiros, porém, essa implementação apresenta algumas falhas.
5     * Utilize o sistema de depuração automática para encontrar essas falhas.
6     *
7     * Dica:  as falhas não estão nas declarações de variáveis ou passagem de
8     * parâmetros.
9     *
10    *
11    * Esse método recebe como entrada três números inteiros e
12    * devolve a mediana de C.
13    */
14    public int encontraMediana(int a, int b, int c) {
15      int resultado;
16      if (a < b) {
17        if (b < c) {
18          if (a < c)
19            resultado = c;
20          else
21            resultado = a;
22        } else {
23          resultado = b;
24        }
25      } else {
26        if (c > b)
27          resultado = b;
28        else if (c > a)
29          resultado = a;
30        else
31          resultado = c;
32      }
33      return resultado;
34    }
```

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 != 0)
    resultado = n/2;
  else
    resultado = (n-3)/2;
  return resultado
}
```

(a) P1a

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 == 0)
    resultado = n/2;
  else
    resultado = (n-2)/2;
  return resultado
}
```

(b) P1b

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 != 0)
    resultado = n/2;
  else
    resultado = n/2;
  return resultado
}
```

(c) P1c

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 != 1)
    resultado = (n-3)/2;
  else
    resultado = n/2;
  return resultado
}
```

(d) P1d

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 == 0)
    resultado = n/2;
  else
    resultado = n/2;
  return resultado
}
```

(e) P1e

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 == 0)
    resultado = n/2;
  else
    resultado = (n-3)/2;
  return resultado
}
```

(f) P1f

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 != 0)
    resultado = (n-3)/2;
  else
    resultado = n/2;
  return resultado
}
```

(g) P1g

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 == 1)
    resultado = (n-3)/2;
  else
    resultado = n/2;
  return resultado
}
```

(h) P1h

```
public int calcula(int n) {
  int resultado = 0;
  if (n % 2 != 1)
    resultado = n/2;
  else
    resultado = (n-3)/2;
  return resultado
}
```

(i) P1i

Tabela 8.4: Programas construídos por alunos na tentativa de solucionar o Problema P1.

contém as seguintes falhas: o operador “<” da linha 17 deveria ser “>” e o operador “>” da linha 26 deveria ser “<”. Os alunos deverão modificar esse programa para fazer com que todos os casos de teste fornecidos sejam executados com sucesso.

Como descrito na justificativa da Tabela 8.6, esse problema tem como objetivo explorar o recurso de refinamento de componentes abstratos na análise de um programa com aninhamentos do comando *if-then-else*. Ou seja, permitir que o aluno lide com o problema da *correção dual para if-then-else* quando esses comandos estiverem aninhados de forma complexa (o Programa 5 possui 5 comandos *if-then-else* com 3 níveis de aninhamento para cada parte do comando mais externo). Note que o depurador é capaz de fazer com que o aluno se concentre apenas em uma fração das linhas do programa (no caso do Programa 5, em apenas $\frac{1}{2^3}$ das linhas do programa que estão envolvidas na falha para um caso de teste selecionado). Isso porque, a cada nível de aninhamento

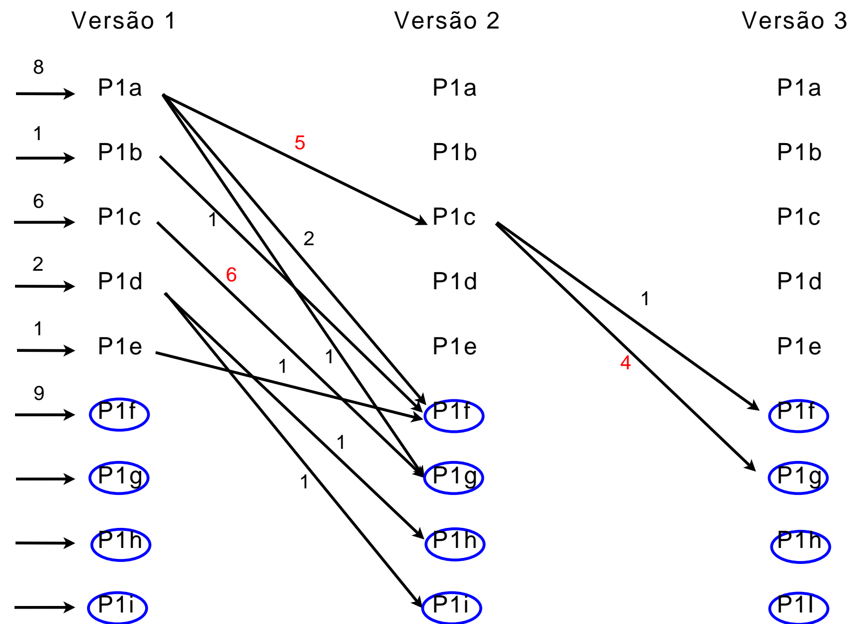


Figura 8.1: Sequência de modificações nos programas utilizados pelos alunos durante a solução do Problema P1, que conseguiram construir um programa correto. O circulo em volta dos identificadores de programas denotam que aquele programa está correto.

o depurador aponta falhas somente nos blocos executados.

Foram analisados os logs de 16 alunos, sendo que 12 alunos usaram o depurador enquanto 4 não usaram. Dos 4 que não usaram apenas 2 conseguiram resolver o problema porém, modificando completamente o Programa 5 dado pelo professor. Dos 12 alunos que usaram o depurador, 7 conseguiram resolver o problema corretamente realizando modificações exatamente nas linhas apontadas como hipóteses de falha pelo depurador para os casos de teste selecionados.

A Tabela 8.7 mostra os 4 programas construídos pelos alunos que conseguiram resolver o problema corretamente e que usaram o depurador (descritos em termos de modificações feitas no Programa 5, por motivo de espaço). A Figura 8.2 mostra o número de alunos que escolheram as mesmas sequências de modificações de programas durante a resolução do problema, e obtiveram sucesso. Com base nessa figura, podemos observar que:

- 3 alunos usaram o caso de teste CT1 e corrigiram a condição da linha 17 (falha apontada pelo depurador após refinar 2 vezes a hipótese inicial de falha) indo para o programa P2b. Em seguida esses 3 alunos usaram o caso de teste CT3 e modificaram a condição da linha 26 indo para o programa P2d (falha apontada pelo depurador após refinar 2 vezes a hipótese inicial de falha);
- 1 aluno usou o caso de teste CT1 e fez as duas correções que os 3 alunos acima fizeram mas em apenas um passo da chamada do depurador resultando no programa P2d. Isso indica que além de seguir a indicação de falha do depurador, esse aluno observou que outros testes

1	P1a		
2	P1a → CT2 → CT1	S	
3	P1a → CT2 → P1f	S	C
4	P1b → CT2 → P1f	S	C
5	P1c → CT2 → P1g	S	C
6	P1c → CT2 → P1g	S	C
7	P1c → CT2 → P1g	S	C
8	P1c → CT2 → P1g	S	C
9	P1a → CT1 → P1c → CT4 → P1g	S	C
10	P1f		C
11	P1f		C
12	P1a → CT1 → P1f	S	C
13	P1a → CT1 → P1c → CT4 → P1g	S	C
14	P1a → CT1 → P1c → CT4 → P1g	S	C
15	P1f		C
16	P1a → CT1 → P1c → CT4 → P1g	S	C
17	P1f		C
18	P1f		C
19	P1d → P1h		C
20	P1f		C
21	P1f		C
22	P1a → CT1 → P1g	S	C
23	P1f		C
24	P1a → CT1 → P1c	S	
25	P1c → CT2 → P1g	S	C
26	P1d → CT1 → P1i	S	C
27	P1f		C
28	P1c → CT2 → P1g	S	C
29	P1e → CT2 → P1f	S	C
30	P1a → CT1 → P1c → CT2 → P1f	S	C

Tabela 8.5: Sequência de programas construídos pelos alunos e casos de testes usados na depuração para solucionar o Problema P1. “S” indica alunos que usaram o depurador e “C” indica alunos que conseguiram gerar um programa correto.

que falharam para o programa dado também indicavam falha no bloco *else* do comando mais externo;

- 1 aluno usou o caso de teste 1 e apenas com um refinamento decidiu corrigir a condição da linha 16 chegando no programa P2a, o que fez com que o CT1 fosse executado com sucesso. Em seguida, o aluno usou o caso de teste CT2 e modificou o programa para P2d;

- 2 alunos também corrigiram a falha apontada pelo teste CT1 porém, ao invés de modificar a condição, esses alunos modificaram o bloco *then* (Linha 19, apontada como falha) e também o *else* (Linha 23, não apontada como falha), porém isso fez com que o teste CT1 obtivesse sucesso; em seguida esses alunos usaram o teste CT3, desfizeram as modificações anteriores e decidiram modificar as 2 condições como no caso anterior, indo para o programa P2d.

É interessante notar que, para o Problema P2, os alunos que souberam corrigir o programa

<table>
<tr><td>Nome:</td><td colspan="3">Cálculo da mediana</td></tr>
<tr><td>Enunciado:</td><td colspan="3">Dados 3 números inteiros, determinar um inteiro n, tal que $n_i \leq n \leq n_j$, sendo n_i e n_j os outros inteiros dados (lidos). Encontre e corrija os erros no programa fornecido (Programa 5).</td></tr>
<tr><td>Tipo:</td><td colspan="3">Modifique</td></tr>
<tr><td rowspan="13">Casos de teste:</td><td>Identificador</td><td>Entradas</td><td>Saídas</td></tr>
<tr><td>CT1</td><td>1, 2, 3</td><td>2</td></tr>
<tr><td>CT2</td><td>1, 3, 2</td><td>2</td></tr>
<tr><td>CT3</td><td>2, 1, 3</td><td>2</td></tr>
<tr><td>CT4</td><td>2, 3, 1</td><td>2</td></tr>
<tr><td>CT5</td><td>3, 1, 2</td><td>2</td></tr>
<tr><td>CT6</td><td>3, 2, 1</td><td>2</td></tr>
<tr><td>CT7</td><td>9, -100, 10</td><td>9</td></tr>
<tr><td>CT8</td><td>12, 12, 5</td><td>12</td></tr>
<tr><td>CT9</td><td>12, 5, 12</td><td>12</td></tr>
<tr><td>CT10</td><td>5, 12, 12</td><td>12</td></tr>
<tr><td>CT11</td><td>12, 12, 12</td><td>12</td></tr>
<tr><td>CT12</td><td>0, -1, 1</td><td>0</td></tr>
<tr><td>Classe:</td><td colspan="3">Mediana</td></tr>
<tr><td>Justificativa:</td><td colspan="3">Esse problema tem como objetivo fixar os conceitos de padrões de seleção. O problema também é um bom exercício para utilizar a funcionalidade de refinamento de componentes abstratos do Dr. Java Pro, visto que cada seleção aninhada é representada por um componente abstrato.</td></tr>
</table>

Tabela 8.6: Atributos do Problema P2.

usando informações dadas pelo depurador, decidiram modificar as linhas do programa (hipóteses de falha) indicadas após um ou mais refinamentos de hipóteses mais abstratas. Isso indica que o diagnóstico hierárquico pode ajudar o aluno na compreensão e correção de falhas em programas mais complexos, como o Programa 5, que contém vários comandos `if-then-else` aninhados.

P2a	Modificação da condição: troca do operador da linha 16 de “<” para “>”.
P2b	Modificação da condição: troca do operador da linha 17 de “<” para “>”.
P2c	Inversão dos blocos: Linhas 19 e 23.
P2d	[Programa correto]. Modificação da condição: troca do operador “<” da Linha 17 para “>” e troca do operador “>” da linha 26 para “<”.

Tabela 8.7: Programas construídos por alunos que resolveram o Problema P2 corretamente, descritos em termos de modificações feitas no Programa 5, dado pelo professor.

8.3 Avaliação do Dr. Java Pro realizada pelos alunos

Para obter uma avaliação da ferramenta Dr. Java Pro por parte dos alunos, preparamos um questionário, mostrado no Apêndice D, contendo treze questões que os alunos responderam após terem passado pelas duas aulas de avaliação. As questões foram divididas em dois grupos:

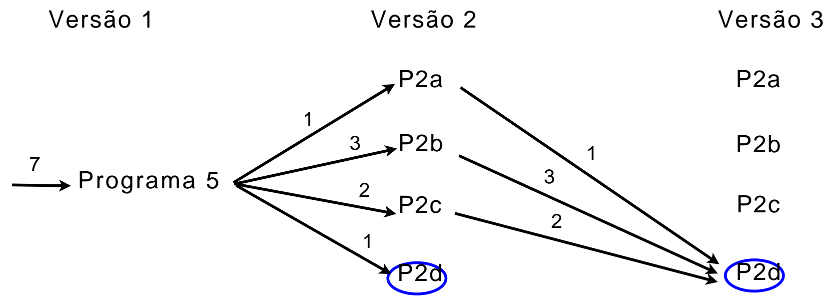


Figura 8.2: Sequência de modificações nos programas utilizados pelos alunos para solucionar o Problema P2.

- *avaliação das aulas no laboratório*, que contém questões sobre a organização do laboratório, equipamento, problemas utilizados nas aulas, didática do professor e sobre o impacto desse tipo de aula no aprendizado de cada aluno.

- *avaliação da ferramenta de Depuração Automática*, que contém perguntas mais específicas sobre a ferramenta, como por exemplo: sobre a familiaridade dos alunos com a ferramenta, sobre a familiaridade a respeito de funcionalidades específicas do depurador automático, casos de teste, padrões elementares, hipóteses de falha, entre outras.

No total, 39 alunos responderam ao questionário, sendo que todos os alunos estavam presentes em pelo menos uma das aulas práticas de avaliação. As respostas relacionadas às aulas práticas no laboratório apontaram que a maioria dos alunos considerou as instalações do laboratório como sendo regular e boa. Alguns problemas como: configuração de máquina, indisponibilidade de máquinas, entre alguns outros problemas menos importantes, provavelmente influenciaram essas repostas. Segundo os alunos, a escolha dos problemas selecionados, assim como a didática do professor foram consideradas boas (dentro das alternativas péssima, ruim, regular, boa e ótima) .

A maioria dos alunos (97%) achou interessante a aula prática no laboratório e acreditou que esse tipo de aula contribuiu para o seu aprendizado. Talvez, por essa razão, 95% dos alunos gostariam de ter outras aulas práticas, caso tivessem que cursar outra disciplina de programação.

Questões sobre a ferramenta de depuração automática. No início da disciplina, foi elaborado um tutorial de uso da ferramenta, que ficou disponível para os alunos até o final da disciplina. O objetivo desse tutorial foi auxiliar os alunos na instalação e utilização do Dr. Java Pro. Apesar disso, somente 31% dos alunos confirmaram terem lido o tutorial. Como o Dr. Java Pro não foi adotado pelo ministrante como a ferramenta da disciplina, e sim o Dr. Java original, os alunos não tiveram interesse em estudar o seu funcionamento antes das aulas práticas de avaliação. Porém, a maioria dos alunos (72%) considerou, após a aula prática de avaliação, ter adquirido um nível médio de familiaridade com a ferramenta Dr. Java Pro, o que indica que as modificações feitas no Dr. Java original são de fácil uso e compreensão.

De uma forma geral, 70% dos alunos gostaram dos novos recursos fornecidos pela ferramenta:

98% acharam interessante o fato dos problemas já estarem disponíveis na ferramenta; 70% acharam interessante e útil as indicações do depurador em termos de hipóteses de falha; e 100% dos alunos acharam realmente interessante ou gostariam de ter explorado mais o uso de casos de teste para auxiliar no desenvolvimento do programas (esse comportamento foi percebido já na primeira aula de avaliação, no qual os alunos tiveram o primeiro contato com a ferramenta.

Problema	P1	P2
Participantes	30	16
Interações com o sistema	3401	3909
Casos de teste do problema	4	12
Execuções dos casos de teste	202	187
Chamadas ao depurador automático	341	203
Valores informados para variáveis	193	94
Refinamentos	86	68
Participantes que acertaram o problema	27 (90,0%)	9 (56,25%)

Tabela 8.8: Número de interações dos alunos com o Dr. Java Pro.

Quanto aos padrões elementares, vistos como componentes abstratos no programa do aluno, apesar de termos ministrado uma aula teórica falando sobre esses padrões, a maioria dos alunos respondeu ter pouca familiaridade com esse conceito. Acreditamos que se mais aulas teóricas forem ministradas usando os padrões de programação implementados no Dr. Java Pro, melhor será o auxílio que o depurador hierárquico poderá fornecer ao aluno. Os resultados extraídos dos *logs* a respeito do problema da mediana (Problema P2), cujos aninhamentos de comandos de seleção são tratados em vários níveis de abstração pelo depurador, refletem essa dificuldade. No entanto, no questionário, a grande maioria dos alunos (98%) disseram ter compreendido as funcionalidades de refinamento de componente abstrato e como informar valores para variáveis.

8.4 Conclusão das Análises Experimentais

A avaliação do Dr. Java Pro através da análise dos logs dos alunos nas aulas práticas de avaliação para os 2 problemas selecionados, P1 e P2, mostraram que a maioria dos alunos acertaram os problemas propostos. A Tabela 8.8 mostra esse resultados considerando todos os alunos que participaram da avaliação dos dois problemas propostos. Essa tabela mostra ainda o número total de interações com o sistema, de execuções de casos de teste, de chamadas ao depurador automático, de refinamentos de hipóteses abstratas e o número de vezes que o aluno informou valores para as variáveis.

É interessante notar que, com base na correção de provas de alunos da mesma disciplina de Introdução à Programação, ministrada em semestres anteriores, sabe-se que na média apenas 30% dos alunos conseguem acertar integralmente uma questão em prova escrita que seja equivalente ao Problema P2. Assim, com base nas porcentagens de alunos que acertaram os dois problemas, 90% e 56,25% respectivamente (Tabela 8.8), podemos sugerir que o uso da ferramenta Dr. Java Pro

auxiliou os alunos na detecção e correção dos problemas de programação propostos.

As análises realizadas na Seção 8.2 mostraram que os alunos que usaram as informações dadas pelo depurador automático para um caso de teste selecionado, de fato corrigiram seus programas com base nessas informações. Além disso, os alunos que usaram o depurador e acertaram o problema, correspondem a maioria dos alunos que acertaram os dois exercícios (63% e 78%, respectivamente).

Com base na avaliação dos alunos sobre a ferramenta Dr. Java Pro os alunos acreditam que a ferramenta é útil para o aprendizado e que ela deveria ser usada mais frequentemente em cursos de Introdução à Programação.

Capítulo 9

Conclusão e Trabalhos Futuros

Um fator importante que torna difícil a aprendizagem de programação é que alunos novatos não possuem conhecimento prévio sobre resolução de problemas de programação e, por essa razão, não conseguem buscar soluções análogas para solucionar um novo problema. Para tentar resolver esse problema, vários educadores utilizam uma metodologia de ensino baseada na resolução de problemas, na qual vários problemas são solucionados em sala de aula através da simulação da execução dos programas. Para dar suporte a essa abordagem, também são utilizados os padrões elementares, que, além de prover soluções prontas para os alunos também permitem ajudar a ensinar boas práticas de programação.

Outra técnica adotada pelos educadores é o processo de depuração, que apesar de ser utilizado para encontrar falhas em programas, também pode auxiliar na aprendizagem de programação. Na depuração, o aluno é obrigado a entender vários detalhes sobre o comportamento das instruções da linguagem, mas principalmente sobre o comportamento do seu código. A depuração é normalmente feita em sala de aula através de simulações com lápis e papel.

Nessa dissertação foi proposta uma ferramenta de depuração automática de programas que é capaz de encontrar falhas em programas de alunos utilizando a técnica da Inteligência Artificial conhecida por Diagnóstico Baseado em Modelos (MBD) com uma extensão para diagnóstico hierárquico, que permite ao aluno depurar o programa em diversos níveis de abstração. Isto é feito transformando laços, seleções, métodos e os padrões elementares reconhecidos no programa em componentes abstratos. O aluno pode então visualizar seu programa em função desses componentes abstratos ou então detalhar esses componentes e visualizá-lo com um nível maior de detalhamento.

O depurador automático de programas foi implementado e integrado à ferramenta Dr. Java, internacionalmente usada em cursos de Introdução à Programação na linguagem Java. Chamamos essa ferramenta de **Dr. Java Pro** (Dr. Java Professor). A implementação do depurador automático foi verificada com relação a um conjunto de 19 problemas de teste.

Foi feita ainda uma avaliação preliminar da ferramenta proposta quanto ao seu uso por um grupo de alunos de uma disciplina de Introdução à Programação. Essa avaliação (Seção 8.2)

mostrou que os alunos que usaram as informações dadas pelo depurador automático para um caso de teste selecionado, de fato corrigiram seus programas com base nessas informações. Além disso, os alunos que usaram o depurador e acertaram o problema, correspondem a maioria dos alunos que acertaram os dois problemas (63% e 78%, respectivamente) usados na avaliação.

Acreditamos que, para uma análise mais detalhada do Dr. Java Pro, será necessário que a ferramenta seja adotada oficialmente em uma disciplina de Introdução à Programação e que nessa disciplina sejam apresentados, em sala de aula, os conceitos de padrões elementares. Para isso, será preciso estender a linguagem reconhecida pelo analisador sintático do depurador automático.

9.1 Principais contribuições

Esse trabalho contribuiu, principalmente, com a especificação de uma nova técnica de depuração automática de programas utilizando níveis de abstração, que tanto pode ser aplicada no processo de aprendizagem de programação quanto em outros domínios de diagnóstico automático. Além disso, apresentamos uma ferramenta, chamada de Dr. Java Pro, que implementa essa nova técnica no ensino de programação na linguagem Java integrada ao ambiente de programação Dr. Java. Mais detalhadamente, as contribuições dessa dissertação foram:

- Fizemos uma revisão dos principais conceitos de diagnóstico baseado em modelos, que inclui vários detalhes sobre as tarefas de geração de hipóteses de falhas e discriminação de hipóteses. Além disso, também fizemos uma revisão do algoritmo *Minimal Hitting Set* (conhecido como algoritmo de Reiter), que apesar de ser um algoritmo geral para a determinação de conjuntos de cortes minimais, é muito utilizado na área de diagnóstico automático e revisão de crenças.
- Detalhamos uma técnica de propagação de restrições utilizada para gerar o conjunto de contribuintes (não facilmente encontrada na literatura da área).
- Fizemos uma revisão da técnica de depuração de programas baseada em modelos, que pode ser utilizada na tarefa de depuração de programas e também no processo de aprendizagem de programação.
- Apresentamos uma revisão da literatura existente sobre a técnica de diagnóstico hierárquico baseado em modelos.
- Apresentamos um novo algoritmo de diagnóstico hierárquico baseado em modelos para ser usado na depuração automática de programas de alunos [33, 32], mas que também pode ser usado em outras aplicações de diagnóstico hierárquico.
- Implementamos em Java o sistema de depuração hierárquica de programas apresentado nesse trabalho e também integramos esse sistema com uma ferramenta de programação, que foi batizada como Dr. Java Pro [34]. A versão final dessa ferramenta ficará disponível ao público através de uma licença de código aberto.

- Disponibilizamos na Internet um tutorial sobre o uso das principais funcionalidades do Dr. Java Pro, relacionadas à depuração automática.
- Fizemos uma avaliação preliminar da ferramenta de depuração automática através de experimentos realizados com alunos de uma disciplina de Introdução à Programação.

9.2 Trabalhos futuros

Entre as possibilidades de estender esse trabalho, consideramos os seguintes pontos como sendo mais importantes: *implementação dos algoritmos*, *técnicas de diagnóstico e depuração automática* e *avaliação pedagógica*, conforme descritos a seguir.

9.2.1 Implementação

Apesar da nossa implementação permitir que sejam cobertos vários assuntos abordados em uma disciplina de Introdução à Programação, acreditamos que é interessante que no futuro a ferramenta dê suporte aos seguintes aspectos adicionais:

- **Recursão.** Apesar desse assunto não ser, em geral, abordado em disciplinas de Introdução à Programação, é interessante desenvolver um mecanismo que possibilite modelar e tratar esse tipo de construção;
- **Uso de múltiplas classes.** Apesar da nossa implementação ser capaz de construir modelos com vários métodos, é interessante também utilizar várias classes (algumas disciplinas de Introdução à Programação utilizam várias classes para realizar uma tarefa complexa ou simplesmente para obter um certo reaproveitamento). Outra questão é que diversos padrões utilizam múltiplas classes. Por essa razão, acreditamos que seja interessante implementar essa funcionalidade.
- **Vetores e Matrizes**. A aprendizagem de vetores e matrizes requer que o aluno utilize diversas técnicas e estratégias aprendidas no decorrer da disciplina. Além disso, isso aumenta consideravelmente o número de problemas que podem ser utilizados.
- **Uso de objetos**. Utilizando um modelo do programa capaz de representar objetos possibilita que o depurador seja capaz de tratar diversas situações, inclusive em programas mais avançados.

9.2.2 Técnicas de diagnóstico e depuração automática

As seguintes extensões relacionadas às técnicas de diagnóstico baseado em modelos podem ser feitas no nosso trabalho:

- **Uso de modelos de falhas**. Um modelo de falhas contém informações a respeito do comportamento dos componentes do sistema quando eles estão falhos [40]. Num depurador automático, essas informações de falha podem ser utilizadas para informar como certos tipos de instruções podem ser implementadas por alunos aprendizes.

- **Modelar as variáveis do sistema**. Em [12], é apresentada uma abordagem que possibilita o sistema de diagnóstico a encontrar falhas em conexões do sistema. Visto que no depurador automático as conexões estão relacionadas aos valores das variáveis em um determinado ponto do programa, esse informação pode ser útil para detectar situações em que um valor incorreto foi atribuído para uma variável.

- **Explorar falhas estruturais**. Como citado nesse trabalho, a depuração automática de programas feita com técnicas de diagnóstico baseado em modelos é capaz de encontrar somente alguns tipos de falhas estruturais. Por essa razão, seria interessante incorporar informações no modelo (ou heurísticas ao depurador) que permitissem encontrar certos tipos de falhas estruturais.

- **Uso de modelos com probabilidades de falhas**. Um modelo com probabilidades de falhas indica a probabilidade na qual cada componente de um sistema físico pode apresentar falha [13]. Da mesmo forma, no caso da depuração automática de programas, esse modelo pode indicar a probabilidade de falhas de programação que alunos podem cometer.

Além dessas extensões citadas, também é possível utilizar o depurador automático de programas como parte integrante de um Sistema Tutor Inteligente. Um **Sistema Tutor Inteligente** (*Intelligent Tutoring System* - ITS) [45] é uma ferramenta de aprendizado eletrônico (*e-learning*) que usa técnicas de Inteligência Artificial a fim de fornecer subsídio adaptado ao perfil do estudante. Um aspecto importante em um ITS é a forma como representar o conhecimento do aluno, conhecido como *modelo do aluno* ou *diagnóstico do aluno* [37] que é, em geral, construído com base na avaliação das soluções de problemas propostos ao aluno. Baseado no modelo do aluno, um ITS é capaz de tomar decisões instrucionais, por exemplo, quando mudar para um novo tópico ou que tipo de atividade propor para um determinado aluno. Utilizando o depurador automático de programas é possível construir o modelo do aluno com base nas hipóteses de falha geradas para o programa construído pelo aluno, para solucionar um determinado problema.

9.2.3 Avaliação pedagógica

Uma tarefa importante para concluir a avaliação e a aceitação da ferramenta proposta seria utilizá-la durante um semestre inteiro com uma turma da disciplina de Introdução à Programação.

Apesar de termos feito uma avaliação preliminar com alunos em duas aulas práticas, o fato da ferramenta não ter sido utilizada com mais frequência pelos alunos impossibilitou uma avaliação mais precisa. Isso aconteceu porque em vários momentos os alunos tinham dúvida a respeito

dos conceitos de depuração automática. Mesmo tendo disponibilizado um tutorial da ferramenta, os alunos não tiveram tanto interesse em realmente compreender o processo de depuração e as funcionalidades disponíveis.

Também é importante avaliar a ferramenta com uma turma na qual os alunos aprendam a programar utilizando padrões elementares. Esse tipo de aprendizado é importante, porque introduz o aluno aos conceitos de abstrações e métodos de resolução de problemas, que são pontos centrais de nossa ferramenta.

Por fim, seria interessante fazer uma avaliação que comparasse alunos de uma turma que usaram a ferramenta contra a alunos de outra turma que não utilizaram a ferramenta. Dessa forma, será possível dizer se o uso de uma ferramenta de depuração automática de programas realmente pode auxiliar alunos no processo de aprendizagem de programação.

Parte IV

Apêndices

Apêndice A

Padrões elementares

Nesse apêndice, descrevemos os padrões elementares citados no decorrer desse trabalho.

Seleção simples

Um conjunto de ações deve ser executada ou não, dependendo de alguma condição testável.

Por exemplo, ao final de um processamento se o número obtido em uma operação for maior que zero, então deverá ser enviada uma mensagem de alerta para o usuário.

```
if (resultado > 0) {
  enviarAlerta();
}
```

A ação não precisa ser repetida, somente ser executada uma única vez. Também não existem condições que devem ser executadas exclusivamente quando a condição for falsa.

Nesse caso, deve ser utilizado uma instrução `if` sem a parte `else`.

```
if (<condição>)
  ações;
```

Observe que as `ações` somente serão executadas caso a `condição` seja verdadeira.

Seleção alternativa

Um dentre dois conjuntos possíveis de ações deve ser executado, dependendo de uma condição. Quando a condição for verdadeira, deverá ser executado um conjunto de ações, mas quando a condição for false, deverá ser executado um outro conjunto de ações.

Por exemplo, ao final de um processamento se o número obtido em uma operação for igual a

zero, deverá ser apresentada uma mensagem de sucesso para o usuário do sistema, caso contrário deverá ser emitia uma mensagem de alerta para o usuário.

```
if (resultado > 0) {
  enviarMensagemSucesso();
} else {
  enviarAlerta();
}
```

Assim como no caso do padrão Seleção simples, a ação precisa ser executada uma única vez. Porém, nesse caso, deve ser executado um conjunto de ações quando a condição for verdadeira e outro conjunto de ações quando a condição for falsa.

Nessa situação, utilize uma instrução `if` com a parte `else`.

```
if (<condição>)
  ações1;
else
  ações2;
```

Repetição contada

Um conjunto de ações deve ser executado para cada elementos de uma coleção. O número de elementos na coleção é conhecido antes do início do processamento.

Por exemplo, a soma de todos os números inteiros de um conjunto C deve ser computada. O conjunto tem tamanho n e cada elemento desse conjunto é acessível por $c[k]$, onde k é um índice de C, variando entre 0 e $n-1$.

```
soma = 0;
for (i = 0; i < n; i++) {
  soma = soma + c[i];
}
```

Cada elemento de um conjunto precisa ser processado única vez e todos os elementos desse conjunto precisam ser processados.

Nesse caso, utilize uma laço do tipo `for`, indexado por uma variável que começa em zero (ou no índice do primeiro elemento válido da coleção) e vai até o último elemento válido da coleção. Nessas condições, as ações no corpo do laço serão executadas para cada elemento da coleção.

```
for (<inicialização do indexador>; <condição>; <atualiza o indexador>)
  ações;
```

Note que esse tipo de processamento também pode ser feito com um laço do tipo `while`, com a inicialização do indexador antes do início do laço e a atualização do indexador deve ocorrer dentro do laço.

Repetição com sentinela

Um conjunto de ações deve ser executado para cada elementos de uma coleção, até que um elemento com valor específico seja encontrado. O tamanho da coleção não é conhecido antes do processamento, ou ela pode ser gerada dinamicamente.

Por exemplo, a soma de todos os números inteiros positivos de um conjunto C deve ser computada. Todos os elementos dessa coleção devem ser somados e o processamenteo termina quando o valor especial -1 for encontrado. Cada elemento desse conjunto C é acessível por $c[k]$, onde k é um índice de C, variando entre 0 e a posição onde se encontra o elemento com valor -1.

```
soma = 0;
i = 0;
while (c[i] != -1) {
  soma = soma + c[i];
i = i + 1;
}
```

Cada elemento de um conjunto precisa ser processado única vez e todos os elementos desse conjunto precisam ser processados até que seja encontrado um determinado valor especial.

Nessa situação, pode ser utilizado um laço do tipo `while`, indexado por uma variável iniciando em zero (ou no índice do primeiro elemento válido da coleção) e segue o processamento enquanto não for encontrado um elemento na posição corrente do indexador com um valor especial, que indica fim do processmanto. Além disso a variável utilizada para indexar a coleção deve ser inicializada antes do laço e ter seu valor atualizado dentro do laço.

```
<inicialização do indexador>;
while(c[<variável indexadora] != <valor de final de processamento>) {
  ações;
  <atualiza o indexador>;
}
```

Apêndice B

Problemas de teste

Nesse apêndice, apresentamos uma breve descrição de cada problema de teste utilizado para fazer a verificação de implementação do Dr. Java Pro, citada no Capítulo 6.

Maratona

Dados três inteiros, h, m e s, representando o tempo consumido por um atleta em uma maratona, em horas, minutos e segundos; outro inteiro m representando a distância percorrida pelo atleta em metros, e mais um inteiro kg representando o peso do atleta em quilos no final da corrida. Encontre a velocidade média desse atleta em metros por segundo e calcule o peso transformado em gramas.

Max

Dados dois números inteiros, a e b, calcule o valor do maior desses números multiplicado por 3.

CalculaPremio

Dados dois inteiros, *posicao* e *premio*, calcule o valor final do prêmio de um participante em uma competição, conforme os critérios a seguir:

- se a posição do participante (variável *posicao*) for igual à 1, o valor final do prêmio é dado pela seguinte fórmula: $\frac{premio \times 12 \times premio}{100} + 50$;
- se a posição do participante for maior que 1, valor final do prêmio é calculado pela fórmula: $\frac{premio \times 7 \times premio}{100}$;

VerificaPar

Dado um número inteiro n, verifique se n é par.

Mediana

Dados três número inteiros, a, b e c, encontre a mediana do conjunto formado por a, b e c.

Multa

Dados três números inteiros: (1) um inteiro representando o valor de um pagamento, (2) um inteiro 1 ou 0 indicando se deverá ser aplicada uma multa por atraso no pagamento e; (3) outro inteiro 1 ou 0 indicando se deverá ser aplicada uma multa de mora. Calcule o valor final da parcela, da seguinte forma: se a multa por atraso for igual 0, decremente 25 do valor de pagamento, senão incremente 50 no valor de pagamento. Além disso, se a multa de mora for igual a 0, decremente 5 do valor resultante do cálculo da multa por atraso, caso contrário incremente 15 no valor resultante do cálculo da multa por atraso

SomaDigitos

Dado um número inteiro n, calcule a soma de todos os dígitos de n.

Triângulo

Dados três número inteiros, x, y e z, verifique se esses números formam as arestas de um triângulo retângulo.

Palindromo

Dado um número inteiro n, verifique se n é um número palíndromo.

SomaNNumerosMain

Dado um número inteiro n e uma sequência S com n números inteiros, encontre a soma de todos os números de S.

Nota: Utilize obrigatóriamente um laço do tipo `while` no seu programa para resolver esse problema.

SomaNNumerosComForMain

Dado um número inteiro n e uma sequência S com n números inteiros, calcule a soma de todos os número de S.

Nota: Utilize obrigatóriamente um laço do tipo `for` no seu programa para resolver esse problema.

Calculadora

Construa um programa para simular uma calculadora simples, que disponibilize as seguintes operações: adição, subtração, multiplicação e divisão. Todas as operações devem ser receber um número inteiro e devolver como resultado outro número inteiro.

Perfeito

Dado um númeor inteiro n, verifique se n é um número perfeito.

SomaNumeros

Dado um número inteiro, calcule a soma de 1 até n

SomaMultiplos

Dado um número inteiro n, calcule a soma de todos os múltiplos de 5 entre 100 e $n - 1$.

MaxDigitosEmSeq

Dado um número inteiro n, maior que zero, encontre o tamanho (comprimento) da maior subsequência (consecutiva) de dígitos iguais deste número.

Maximiza

Dados dois naturais m e n determinar, entre todos os pares de números naturais (x, y) tais que $x < m$ e $y < n$, um par para o qual o valor da expressão $xy - x2 + y$ seja máximo e calcular também esse máximo.

NumeroSegmentos

Dado um número inteiro n e uma sequência de n números inteiros, determinar quantos segmentos de números iguais consecutivos compõem essa sequência.

Apêndice C

Arquivo de configurações de problemas de testes

Nesse apêndice, mostramos um exemplo do arquivo de configuração de execução dos problemas de teste apresentados no Capítulo 8.

A Figura C.1 mostra um arquivo de configuração de um problema de teste, `SomaMultiplos`, no qual as informações estão em um formato do tipo *chave=valor*.

```
# main definitions.
name=Soma Multiplos Test Suite
resourceName=./programs/SomaMultiplos.java
testCases=tc1

# associate the class/methods with the test cases.
method=main

# test case definitions.
tc1.properties=input0, output0
tc1.input0=100
tc1.output0=100
tc1.expected=6, 8, 9, 10, 15, 12, 13, 14
```

Figura C.1: Arquivo de configuração dos casos de teste para testar uma funcionalidade do Dr. Java Pro.

A chave *name* é usada para definir uma descrição para o conjunto de casos de teste. A chave *resourceName* é usada para definir o programa que será testado. A chave *testCases* define os conjuntos de casos de teste que serão usados para testar o programa definido por *resourceName*. O nome do método que será executado no programa é definido pelo valor da chave *method*. Em seguida, definimos os casos de teste do programa solução. A chave *tc1.properties* define o nome dos parâmetros de entrada e saída do caso de teste tc1. O valor *input0* da chave *tc1.properties* indica que o programa recebe um valor pela entrada padrão e *output0* indica que o programa devolve um valor na saída padrão (também é possível passar os valores de entrada via parâmetros de método pelas chaves *argIn0*, *argIn1*,...,*argInN*, sendo N o número de parâmetros definidos no método testado,

e usar a saída devolvida como resultado da execução da instrução `return` no corpo do método através da chave *argOut*). A chave *tc1.input0* define o valor da primeira entrada do caso de teste tc1 e *tc1.output0* define o valor esperado que o programa devolva. Por fim, a chave *tc1.expected* define as hipóteses de falhas que espera-se que o sistema de depuração automática devolva quando tiver sido feita a depuração do programa com as entradas e saídas definidas nesse arquivo de configuração. Cada hipótese é representada por um conjunto de números inteiros (representando as linhas) separados por vírgula e delimitados por "“{" e "}".

Apêndice D

Questionário de Avaliação da Ferramenta

Nesse apêndice, apresentamos o questionário que os alunos usaram para fazer a avaliação da ferramenta.

Todas as questões apresentadas são de resposta obrigatória.

Sobre as aulas práticas no laboratório CEC

Q1. Sobre as aulas práticas no laboratório CEC, como você avaliaria os seguintes itens?

	Péssimo	Ruim	Regular	Bom	Ótimo
Organização (computadores, data show, ar condicionado, infra-estrutura em geral)					
Seleção dos exercícios (os exercícios feitos foram apropriados?)					
Didática do professor (o professor conseguiu explicar o conteúdo da aula)					

Q2. Você acredita que as aulas no laboratório foram úteis para o seu aprendizado?

- Sim
- Não

Q3. Se você tivesse outra disciplina de programação, gostaria de ter mais aulas práticas no laboratório?

- Sim
- Não

Sobre a ferramenta de Depuração Automática

Essa é a ferramenta integrada com o Dr. Java que você usou nas aulas práticas no CEC. Nessa ferramenta você pode testar os programas com casos de teste e utilizar o depurador automático para encontrar os erros de lógica em seus programas.

Q4. Você leu o tutorial da ferramenta de depuração automáitca disponível na página do curso, no link: `http://paca.ime.usp.br/mod/resource/view.php?id=8465`?

- Sim
- Não

Q5. Você tentou usar a ferramenta de depuração automática em sua casa ou no laboratório do seu instituto? Observe que estamos falando da ferramenta de depuração automática e não do Dr. Java original.

- Sim
- Não

Q6. Como você classifica a sua familiaridade com a ferramenta de depuração automáitca?

- Muito pouca
- Pouca
- Média
- Alta
- Muito alta

Q7. Você acredita que o uso da ferramenta nas aulas de exercícios foi útil para o seu aprendizado?

- Sim

- Não

Q8. A ferramenta de depuração automática apresenta alguns recursos adicionais quando comparada ao Dr. Java origianal. Classifique cada um dos recursos abaixo, de acordo com a sua preferência.

	Não sei o que é isso	Não gostei	Parece ser interessante, mas gostaria de explorar um pouco mais	É realmente interessante!
Atividades pré-definidas (as atividades já estão disponíveis na ferramenta)				
Possibilidade de executar casos de teste				
Auxílio de um depurador automáico que mostra as linhas do programa com falha				

Q9. Os termos abaixo foram apresentados e citados várias vezes na aula de exercícios. Como você classifica a sua familiaridade com esses termos?

	Muito pouca	Pouca	Média	Alta	Muito alta
Casos de testes					
Padrões elementares					
Refinamento de Componente Abstrato					
Informar valores para variáveis					
Hipóteses de falha					

Q10. Durantes os exercícios, você chegou a utilizar os recursos de ”Refinamento de componente abstratro”e ”Informar valores para variáveis”

- Sim
- Não

Q11. Caso você tivesse outra disciplina de programação, você gostaria de utilizar a ferramenta novamente?

- Sim
- Não

Q12. De uma forma geral, como você classificaria a ferramenta de depuração automática?

- Péssima
- Ruim
- Regular
- Boa
- Muito Boa

Referências Bibliográficas

[1] O. Astrachan e E. Wallingford. Loop patterns. In *In Proceedings of the 1998 Pattern Languages of Programs (PLoP'98)*, Agosto 1998.

[2] K. Autio e R. Reiter. Structural abstraction in model-based diagnosis. In *In Proceedings of the 13th European Conference on Artificial Intelligence (ECAI'98)*, páginas 269–273, 1998.

[3] L. N. Barros, A. P. S. Mota, K. V. Delgado, e P. M. Matsumoto. A tool for programming learning with pedagogical patterns. In *eclipse '05: Proceedings of the 2005 Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'05) workshop on Eclipse technology eXchange*, páginas 125–129, New York, NY, USA, 2005. ACM Press.

[4] K. Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[5] R. Benjamins. *Problem Solving Methods for Diagnosis.* PhD thesis, University of Amsterdam, 1993.

[6] J. Bergin. Patterns for selection. `http://csis.pace.edu/ bergin/patterns/ Patternsv4.html`. Acessado em: 26/02/2010.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, e M. Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[8] L. Chittaro e R. Ranon. Hierarchical model-based diagnosis based on structural abstraction. *Artificial Intelligence*, 155(1-2):147–182, 2004.

[9] L. Console, G.E. Friedrich, e D.T. Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *In Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, páginas 1494–1499, Chambéry, France, 1993. Morgan Kaufmann.

[10] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24(1-3):347–410, 1984.

[11] J. de Kleer. "a general labeling algorithm for assumption-based truth maintenance". In *Proceedings of the Seventh National Conference on Artificial Intelligence*, páginas 188–192, San Francisco, California, 1988. AAAI Press.

[12] J. de Kleer. Modeling when connections are the problem. In *In Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI'07)*, páginas 310–317, 2007.

[13] J. de Kleer. Modeling when connections are the problem. In *In Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI'07)*, páginas 310–317, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[14] J. de Kleer e B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[15] K. V. Delgado. Diagnóstico baseado em modelos num sistema tutor inteligente para programação com padrões pedagógicos. Dissertação de mestrado, Instituto de Matemática e Estatística, 2005.

[16] M. Ducassé. A pragmatic survey of automated debugging. In *AADEBUG '93: Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, páginas 1–15, London, UK, 1993. Springer-Verlag.

[17] J. P. East, S. R. Thomas, E. Wallingford, W. Beck, e J. Drake. Pattern-based programming instruction. In *In the Proceedings of the ASEE Annual Conference and Exposition*, Washington, DC, USA, June 1996.

[18] A. Feldman. Hierarchical approach to fault diagnosis. Master's thesis, Delft University of Technology, The Netherlands, November 2004.

[19] W. Nejdl G. Friedrich, G. Gottlob. Physical impossibility instead of fault models. In *Proceedings of the $8^t h$ National Conference on Artificial Intelligence*, páginas 331–336. AAAI, 1990.

[20] E. Gamma, R. Helm, R. Johnson, e J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24(1-3):411–436, 1984.

[22] F. Giunchiglia e T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.

[23] R. Greiner, B. A. Smith, e R. W. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[24] IEEE. Ieee standard glossary of software engineering terminology (ansi/ieee std. 610.12-1990). Technical report, IEEE, 1990.

[25] W. L. Johnson e E. Soloway. Proust: Knowledge-based program understanding. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, páginas 369–380, Piscataway, NJ, USA, 1984. IEEE Press.

[26] C. Mateis, M. Stumptner, e F. Wotawa. Debugging of java programs using a model-based approach, 1999.

[27] C. Mateis, M. Stumptner, e F. Wotawa. A value-based diagnosis model for java programs, 2000.

[28] W. Mayer e M. Stumptner. Model-based debugging - state of the art and future challenges. *Electronic Lecture Notes in Theoretical Computer Science*, 171:61–82, May 2007. Workshop on Verification and Debugging (V&D'06).

[29] W. Mayer, M. Stumptner, D. Wieland, e F. Wotawa. Observations and results gained from the jade project. In *Proceedings of the Thirteenth International Workshop on Principles of Diagnosis*, Semmering, Austria, April 2002.

[30] W. Mayer, M. Stumptner, e F. Wotawa. Can AI help to improve debugging substantially? automatic debugging and the jade project. *Journal of the Austrian Society for Artificial Intelligence*, 21(4):18–22, 2002.

[31] I. Mozetič. Hierarchical model-based diagnosis. *Int. J. Man-Mach. Stud.*, 35(3):329–362, 1991.

[32] W. R. Pinheiro e L. N. Barros. Um tutor inteligente para o ensino/aprendizado de programação com técnicas de diagnóstico hierárquico baseado em modelos. In *In Encontro Nacional de Inteligência Artificial - ENIA'09*, páginas 1089–1098, Bento Gonçalvez, 2009.

[33] W. R. Pinheiro, L. N. Barros, e K. V. Delgado. Programming learning: a hierarchical model based diagnosis approach. In *In Proceedings of International Conference on Interactive Computer aided Blended Learning*, Florianópolis, 2008.

[34] W. R. Pinheiro, L. N. Barros, e F. Kon. AAAP: Ambiente de Apoio ao Aprendizado de Programação. In *Workshop de Ambientes de Apoio à Aprendizagem de Algoritmos e Programação*, São Paulo, 2007.

[35] R. Porter e P. Calder. A pattern-based problem-solving process for novice programmers. In *ACE '03: Proceedings of the fifth Australasian conference on Computing education*, páginas 231–238, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[36] R Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[37] J. Self. Model-based cognitive diagnosis. *User Modeling and User-Adapted Interaction*, 3(1):89–106, 1993.

[38] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[39] P. Struss. What's in sd?: Towards a theory of modeling for diagnosis. páginas 419–449, 1992.

[40] P. Struss e O. Dressier. "physical negation": integrating fault models into the general diagnostic engine. In *In Proceedings of the 11th International Joint Conferences on Artificial Intelligence (IJCAI'89)*, páginas 1318–1323, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[41] M. C. Tanner T. Bylander, D. Allemang e J. R. Josephson. The computational complexity of abduction. *Artificial Intelligence*, 49(1-3):25–60, 1991.

[42] F. Vatan. The complexity of the diagnosis problem. Technical Report Technical Support Package NPO-30315, Jet Propulsion Laboratory, 4800 Oak Drive, Pasadena, CA, April 2002.

[43] E. Wallingford. Roundabout, a pattern language for recursive programming, 1997.

[44] E. Wallingford. The elementary patterns home page. `http://cns2.uni.edu/ wallingf/patterns/elementary/`, 2003. Acessado em: 26/02/2010.

[45] E. Wenger. *Artificial intelligence and tutoring systems: Computational and cognitive approaches to the communication of knowledge.* Morgan Kaufmann Press, 1987.

[46] D. Wieland. *Model-Based Debugging of Java Programs Using Dependencies.* Phd thesis, Technische Universität Wien, 2001.

[47] L. E. Winslow. Programming pedagogy – a psychological overview. *SIGCSE Bull.*, 28(3):17–22, 1996.