Diagnóstico baseado em modelos num sistema tutor inteligente para programação com padrões pedagógicos

Karina Valdivia Delgado

Dissertação Apresentada
ao Instituto de Matemática e Estatística da
Universidade de São Paulo
para Obtenção do grau de mestre
em Ciência da Computação

Àrea de Concentração: **Inteligência Artificial**
Orientadora: **Prof. Dra. Leliane Nunes de Barros**

Durante a elaboração deste trabalho a autora recebeu apoio financeiro da CAPES

São Paulo, dezembro de 2005

Diagnóstico baseado em modelos num sistema tutor inteligente para programação com padrões pedagógicos

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Karina Valdivia Delgado e aprovada pela Comissão Julgadora.

São Paulo, janeiro de 2006

Banca Examinadora :

- Profa. Dra. Leliane Nunes de Barros (Orientadora) - IME-USP
- Prof. Dr. Ronaldo Fumio Hashimoto - IME-USP
- Profa. Dra. Ana Cristina Bicharra Garcia - UFF-RJ

Dedicatória

À Karel, Elias e Sara; meus filhos, com amor e gratidão por sua compreensão e carinho.

Agradecimentos

À professora Leliane Nunes de Barros, pela orientação, confiança e amizade que me brindou durante esses anos.

Aos professores do Departamento de Computação, que contribuíram na minha formação.

Agradeço ao IME e à USP pela oportunidade de realização do curso de mestrado nesta Instuitição

À CAPES pelo apoio financeiro para realização desta pesquisa.

À Wolfganf Mayer e Marcos Chaim, pelos comentários relacionados ao processo de diagnóstico.

À Patricia, Ana Paula e Wendel, pelo trabalho desenvolvido no projeto ProPAT.

Agradeço aos meus amigos Clodis e Andréia, pela revisão do trabalho e o mais importante pela amizade.

À minha familia em São Paulo: Christian, Cristian, Eduardo, Fabio, Felix, Jesús, Thiago e Vladi, que compartilharam comigo momentos maravilhosos que nunca esquecerei.

À minha familia em Arequipa e especialmente aos meus pais Dora e Juan, pelo amor e apoio constante durante toda minha vida.

Aos meus filhos: Sara, Elias e Karel, que com suas palavras ternas e amorosas me deram a força necessária para continuar.

À Deus, meu pai celestial, por estar comigo sempre.

Epígrafe

"É melhor tentar e falhar, que preocupar-se e ver a vida passar; é melhor tentar, ainda que em vão, que sentar-se fazendo nada até o final. Eu prefiro na chuva caminhar, que em dias tristes em casa me esconder. Prefiro ser feliz, embora louco, que em conformidade viver ..."

Martin Luther King Junior

(1929-1968)

Resumo

Tutores Inteligentes são sistemas computacionais de ensino/aprendizagem que empregam técnicas de Inteligência Artificial (IA) com o objetivo de promover o aprendizado individualizado. Um dos aspectos centrais de um sistema tutor para o aprendizado de programação é a depuração do programa construído pelo aluno. O resultado desta depuração serve para guiar o sistema tutor em suas futuras decisões instrucionais, enquanto o processo de depuração em si pode ser explorado para promover a aprendizagem. Ou seja, num processo interativo de depuração, é possível fazer com que o aluno aprenda detectando e corrigindo seus próprios erros. Dentre as propostas de depuração automática de programas, a técnica de IA denominada Diagnóstico Baseado em Modelos (MBD), tem apresentado bons resultados para diagnosticar programas escritos por programadores experientes. Como é feito tradicionalmente para sistemas físicos, MBD analisa um modelo de um programa representado na forma de componentes e conexões, onde os componentes correspondem às estruturas lógicas da linguagem de programação e as conexões representam as constantes e os valores de variáveis. Apesar do sucesso de MBD para depuração de programas, nenhuma proposta foi feita incorporando essa técnica em sistemas tutores.

Este trabalho propõe o desenvolvimento de um sistema de diagnóstico do tipo MBD para analisar programas de alunos de cursos introdutórios de programação. O trabalho estende trabalhos anteriores de depuração automática do tipo MBD para programadores experientes, para ser usado como ferramenta de suporte ao aprendizado de programação. O sistema de diagnóstico desenvolvido **ProPAT_deBUG** faz parte de um ambiente de programação com padrões, chamado ProPAT. Nessa ferramenta, enquanto o aluno edita um programa, ele pode acessar e inserir Padrões Elementares no programa com a intenção de satisfazer sub-metas

de um dado problema. Após o programa ser compilado com sucesso, ele é testado para um conjunto de casos de teste e, em caso de falha, o sistema de diagnóstico é chamado para descobrir falhas lógicas funcionais e estruturais. Os Padrões Elementares usados pelo aluno na construção do programa são usados para a comunicação das hipóteses de falha.

Palavras-chave: Inteligência Artificial, Diagnóstico Baseado em Modelos, Sistemas Tutores Inteligentes de Programação, Ferramentas de Depuração Automática.

Abstract

Intelligent Tutoring Systems are learning and educational tools that use techniques of Artificial Intelligence to promote individualized learning. One of the main aspects of a Programming Tutor System is the debugging of the student program. The result of the debugging is used to guide the tutor system in his future instructional decisions, meanwhile the debugging process can be explored to promote the learning, i.e, in the interactive debugging process it is possible that the student learns by detecting and correcting their own errors. The Model Based Diagnosis (MBD) technique has presented good results to diagnosis programs written by experienced programmers. As it is made for traditional man made systems, MBD analysis the model of the program represented by components and connections, where the components correspond to the logical structures of the programming language and the connections represent the constants and variable values. Despite of the success of MBD for debugging programs, this technique has not yet been applied in tutorial systems.

This work proposes the development of the diagnosis system of type MBD to analyze programs of students of a first programming course. This work extends previous works of automatic debugging of type MDB for experienced programmers. The debugging system **ProPAT_deBUG** is part of a programming environment using *Elementary Patterns*, called ProPAT. In this tool, while the students edit their programs, he can select and insert Elementary Patterns in the program with the intention to satisfy subgoals of a programming problem. After successfully compiling the program, it is tested and in the case of failure, the diagnosis system is called.

Keywords: Artificial Intelligence, Model Based Diagnosis, Intelligent Programming Tutor, Debugging and Testing tools.

Índice

1 Introdução **1**

1.1 Motivação 1

1.2 Diagnóstico Baseado em Modelos para Programadores Experientes 3

1.3 Padrões Elementares de Programação 3

1.4 Objetivos 4

1.5 Organização 4

2 O aprendizado de programação e uma classificação de falhas típicas **7**

2.1 Aprendizado de programação 7

2.1.1 Escolha do paradigma de programação para ensino 8

2.2 Terminologia usada em Depuração 8

2.3 Classificação de falhas em software 9

2.3.1 Classificação de falhas pelas consequências 10

2.3.2 Classificação de tipos de falhas 10

2.4 Falhas típicas de um aprendiz de programação 13

2.5 Considerações finais 17

3 Correção Automática de Programas em Tutores Inteligentes e em Depuradores **19**

3.1 Visão Geral de Técnicas de Correção Automática de Programas 20
3.2 Análise das Técnicas de Correção Automática de Programas 22
3.3 Tutores Inteligentes de Programação 22
3.3.1 Exemplos de Tutores Inteligentes de Programação 23
3.3.2 Comparação entre Sistemas Tutores Inteligentes e Sistemas Gerenciadores de Aprendizagem 26
3.4 Considerações finais 27

4 O projeto ProPAT 29
4.1 Ensino de programação baseado em Padrões Elementares 29
4.1.1 Padrões Elementares 30
4.1.2 Vantagens dos Padrões Elementares no processo de aprendizagem 31
4.2 Documentação e Exemplos de Padrões Elementares 31
4.3 ProPAT: um plug-in Eclipse 37
4.3.1 Perspectiva do Aluno 40
4.3.2 Perspectiva do Professor 40
4.3.3 O sistema ProPAT e um paralelo com os sistemas tutores de programação 42
4.4 Considerações finais 43

5 Diagnóstico Baseado em Modelos de Dispositivos Físicos 45
5.1 Modelo Centrado em Componentes 47
5.2 Diagnóstico Baseado em Consistência 48
5.3 Tarefas do processo de diagnóstico 49
5.3.1 Detecção de Sintomas 49
5.3.2 Geração de Hipóteses 49
5.3.3 Discriminação de hipóteses 53
5.4 Exemplo de aplicação do processo de diagnóstico 54
5.5 Sistemas de Diagnóstico Baseado em Modelos da literatura 57

5.6 Diagnóstico hierárquico 59

5.7 Considerações finais 59

6 Diagnóstico Baseado em Modelos de Programas 61

6.1 Projeto JADE: modelos de programas 61

6.2 MBD de programas para programadores aprendizes 63

6.3 O Modelo Baseado em Valor 64

6.3.1 EXPRESSÕES 66

6.3.2 ATRIBUIÇÃO 68

6.3.3 LEITURA E IMPRESSÃO 70

6.3.4 SELEÇÃO 70

6.3.5 WHILE 72

6.3.6 FUNÇÕES do tipo VOID 75

6.3.7 FUNÇÕES que devolvem valores 75

6.3.8 PROPAT_DEBUG: construção do modelo de componentes e conexões 75

6.4 PROPAT_DEBUG: o Diagnóstico Baseado em Modelos 76

6.4.1 Detecção de sintomas do programa 77

6.4.2 Geração de hipóteses do programa 77

6.4.3 Discriminação de hipóteses do programa 80

6.5 Considerações finais 83

7 Exemplo de Diagnóstico de Programa 85

7.1 Detecção de Sintomas 85

7.2 Modelo Estrutural 86

7.3 Modelo Comportamental 89

7.4 Geração de Hipóteses 89

7.5 Considerações finais 93

8 Resultados experimentais **99**

8.1 Metodologia 99

8.1.1 Medidas usadas na Avaliação do Diagnóstico 100

8.2 Análise dos Resultados 106

8.2.1 Falhas múltiplas alternativas 109

8.3 Considerações finais 112

9 Conclusões e Trabalhos Futuros **113**

9.1 Trabalhos Futuros 114

9.1.1 Extensões do **ProPAT_deBUG** 114

9.1.2 Modelos hierárquicos para diagnóstico 115

9.1.3 Ontologia de Padrões Elementares 117

Lista de Figuras

1.1 Diagnóstico Baseado em Modelos para programas visto como a interação de observações (no programa do aluno) e predições (feitas pelo aluno). 2

2.1 Taxonomia de falhas 12

3.1 Taxonomia de estratégias de correção automática 21

3.2 Decomposição de objetivos em sub-objetivos e planos para o problema 1 25

3.3 Planos usados para o problema 1 26

4.1 Padrão Repetição com sentinela 38

4.2 Padrão Repetição com sentinela continuação 39

4.3 Perspectiva do aluno de ProPAT-Pattern View, Pattern Info View e Exercise View 41

4.4 Perspectiva do professor de ProPAT-Pattern View e Pattern Editor View 42

5.1 Diagnóstico Baseado em Modelos de sistemas físicos, visto como a interação de observações e predições. 46

5.2 Decomposição do processo de diagnóstico em 3 tarefas 50

5.3 Algoritmo de Reiter [Rei87] [GSW89]. 53

5.4 Circuito de um alarme de carro com o componente COMP_W falho. 55

5.5 Aplicação do algoritmo de Reiter para o circuito de um alarme de carro 58

6.1 Diagnóstico Baseado em Modelos (DBM) visto como a interação de observações e predições. A parte (a) descreve um modelo conceitual de DBM para sistemas físicos e a parte (b) descreve o DBM de programas. 62

6.2 Principais diferenças entre o DBM de sistemas físicos e o DBM de programas. 63

6.3 Processo de Diagnóstico Baseado em Modelos do projeto JADE 64

6.4 Modelo de estrutural e comportamental para uma **expressão**. 67

6.5 Modelo estrutural e comportamental para uma **atribuição**. 69

6.6 Modelo estrutural e comportamental para uma sentença de **leitura**. 70

6.7 Modelo estrutural e comportamental para uma sentença de **impressão**. 71

6.8 No modelo baseado em valor, um comando de **seleção** (*if* ou *if-else*) é mapeado a um componente *conditional* com dois subsistemas *thenModel* e *elseModel*. O modelo comportamental desse exemplo foi omitido por motivos de espaço. 73

6.9 Modelo estrutural e comportamental para uma sentença **while**. 74

6.10 Algoritmo de criação do modelo de componentes e conexões do programa do aluno. 76

6.11 Processo de diagnóstico do sistema **ProPAT_deBUG** 77

6.12 Algoritmo de diagnóstico do sistema PROPAT_DEBUG 78

6.13 Regras de produção para o componente multiplicação entre inteiros. Essas regras podem ser usadas para propagar valores para frente ou para trás no modelo do programa. 79

6.14 Estimativa das sentenças estarem corretas, baseadas em casos de teste, para o programa que imprime a mediana de 3 valores x, y e z [JHS02] 82

7.1 Modelo estrutural na forma gráfica do Programa *MAX_1*. 88

7.2 Parte do documento do Padrão Elementar Atribuição. 94

7.3 Parte do documento do Padrão Elementar Seleção Alternativa. 97

8.1 Medida F para k=1 e $\alpha = 0.5$. 107

8.2 Medida F para k=2 e $\alpha = 0.5$. 108

8.3 Medida F para k=3 e $\alpha = 0.5$. 109

8.4 Medida F para k=3 e $\alpha = 0.3$.Os pontos pretos indicam programas com falhas lógicas estruturais. 110

8.5 PRE e COB para k=3. 111

9.1 Padrões usados para o problema 1 117

Lista de Tabelas

2.1 Falhas típicas em estruturas simples. 14

2.2 Falhas típicas em estruturas de seleção. 15

2.3 Falhas típicas em estruturas de repetição. 16

4.1 Estrutura de um documento de ***Design Patterns*** 34

4.2 Estrutura de um documento de *Padrões Elementares* 35

4.3 Exemplos de Padrões Elementares 36

4.4 Motivação para os Padrões Elementares da Tabela 4.3 37

4.5 Alguns dos Tutores Inteligentes de Programação mais citados e técnicas que eles implementam para correção automática. 44

5.1 Alguns termos usados em Diagnóstico Baseado em Modelos. 47

6.1 Gramática BNF da linguagem C restrita. Em que Op é um operador binário que pode ser substituído por: $+$, $-$, $*$, $/$, $\%$, $<$, $>$, $<=$, $>=$, $==$, $!=$, $\&\&$ ou $\|$ e *const* e *id* podem ser do tipo inteiro ou booleano. 65

7.1 Casos de teste usados para o problema MAX 85

7.2 Programa para o problema **MAX** com falha funcional na linha 8 (expressão lógica invertida) ou falha estrutural nas linhas 9 e 12 (linhas trocadas) 86

7.3 Conjunto de axiomas A_s que define formalmente o modelo estrutural do programa do aluno. . . 87

7.4 Axiomas do modelo comportamental. 90

7.5 Hipóteses geradas pelo sistema de diagnóstico para o Programa *MAX_1*. 93

7.6 Diálogo para a discriminação das hipóteses para o Programa *MAX_1*. 95

7.7 Diálogo para a discriminação das hipóteses para o Programa *MAX_1* (continuação). 96

8.1 Enunciado dos problemas usados pelo sistema de diagnóstico. 102

8.2 Casos de teste usados para testar o sistema de diagnóstico 103

8.3 Resultados de aplicar o diagnóstico para diferentes programas de alunos. "*"indica a falta do else no condicional ou a falta de uma execução a mais do laço. 104

8.4 Medidas de precisão e cobertura para os diagnósticos dos 28 programas. 105

8.5 Medida-F para os diagnósticos dos 28 programas. 106

CAPÍTULO 1

Introdução

1.1 Motivação

O diagnóstico cognitivo é definido como o processo de inferir o estado cognitivo de uma pessoa a partir de seu desempenho [Sel93]. A palavra diagnóstico na educação é usada para referir-se às atividades pedagógicas que ajudam a coletar e inferir informações sobre o estudante ou suas ações e não como uma mera detecção de erros [Wen87]. Para os Sistemas Tutores Inteligentes (*Intelligent Tutoring Systems - ITS*), o termo *diagnóstico cognitivo* é usado freqüentemente como sinônimo de *modelagem do estudante* para se referir ao processo de construção de uma representação do conhecimento do aluno a partir das evidências fornecidas pelas soluções de problemas [Sel93].

Em um Sistema Tutor Inteligente de Programação, o diagnóstico automático do programa do aluno é essencial e é considerada uma tarefa difícil. Não existem ferramentas eficientes (depuradores automáticos) que detectem falhas semânticos e lógicos em um programa (da mesma forma que os compiladores detectam falhas de sintaxe). A proposta mais comum encontrada em Tutores Inteligentes de Programação envolve reconhecer as intenções, ou identificar os planos do aluno através de uma análise do programa [Wen87]. Isso é feito armazenando-se planos (fragmentos de programas) que atinjam sub-metas de problemas. A principal dificuldade nessa proposta é que podem existir muitas soluções diferentes para um mesmo problema de programação, ou seja, não se tem um único modelo correto do programa para o problema proposto ao aluno que possa ser usado para identificar as falhas do programa do aluno.

Uma outra proposta possível é, ao invés de modelar uma solução correta para o problema, modelar o programa incorreto do aluno e fazer inferências sobre ele. As observações feitas pelo sistema de diagnóstico sobre o modelo do programa são então comparadas com as entradas e saídas esperadas pelo estudante. A partir dessa comparação são determinadas discrepâncias e concordâncias que são usadas para construir hipóteses sobre as partes falhas do programa do aluno [MSW00]. A Figura 1.1 ilustra esse processo, conhecido por *Diagnóstico*

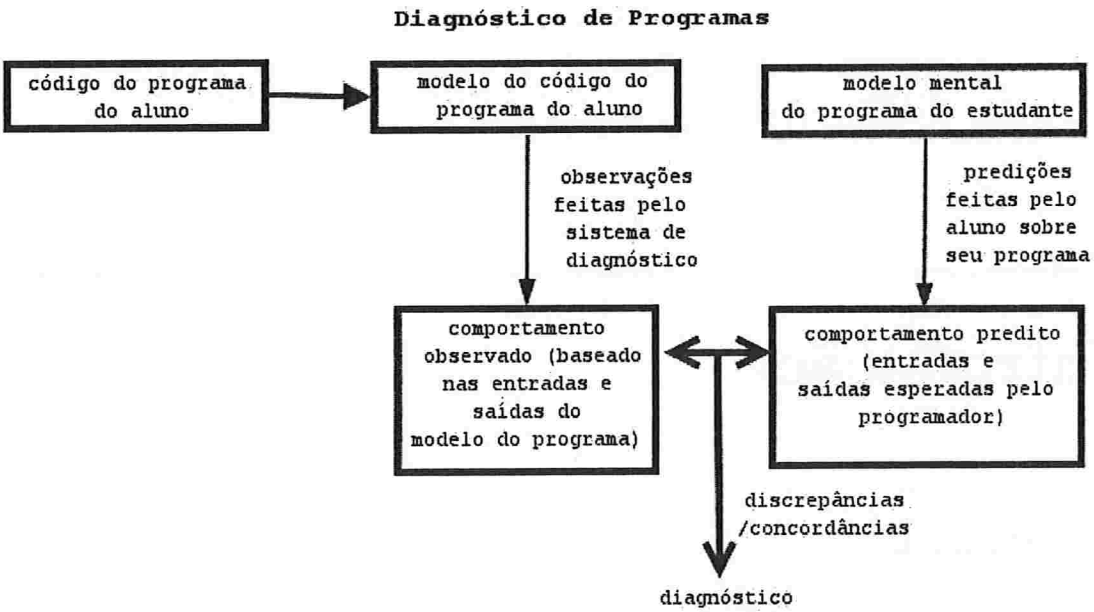


Figura 1.1: Diagnóstico Baseado em Modelos para programas visto como a interação de observações (no programa do aluno) e predições (feitas pelo aluno).

Baseado em Modelos.

Um sistema completo de diagnóstico de programas envolve uma série de atividades, entre elas:

- a construção de um modelo formal do programa do aluno;
- a implementação de métodos para fazer observações de valores de variáveis em diferentes pontos do programa;
- a aquisição de valores previstos pelo aluno para compará-los com as observações;
- a geração de um conjunto de hipóteses de falhas no programa com base nas discrepâncias e concordâncias nos valores das variáveis e
- a discriminação das hipóteses através de novas observações e previsões do aluno.

Ferramentas como a descrita acima têm sido desenvolvidas para programadores experientes (Seção 1.2) mas ainda não foram usadas para fins pedagógicos. Esse trabalho se baseia na premissa de que, dependendo da maneira como as hipóteses de falha de um programa são detectadas e comunicadas ao aluno, um sistema de Diagnóstico Baseado em Modelos pode ser usado como suporte para o aprendizado de programação.

1.2 Diagnóstico Baseado em Modelos para Programadores Experientes

Dentre as propostas mais avançadas de depuração automática de programas, a técnica de Inteligência Artificial (IA) denominada Diagnóstico Baseado em Modelos (Model Based Diagnosis - MBD) [Ben93], tem apresentado bons resultados para diagnosticar programas escritos por programadores experientes [MSW00]. MBD analisa um modelo do programa representado na forma de componentes e conexões, onde os componentes correspondem às estruturas lógicas da linguagem de programação e as conexões, às variáveis do programa.

O modelo de componentes e conexões é usado para propagar os valores de um caso de teste, armazenando as dependendências entre os componentes. As dependências são então usadas para calcular os contribuintes que causaram a discrepância de valores das variáveis no programa. Os contribuintes são transformados em um conjunto de hipóteses usando um algoritmo que determina os conjuntos minimais de componentes que explicam as observações.

Como falamos anteriormente, apesar dos bons resultados [MS+02], as técnicas de MBD para programas não têm sido usadas para fins pedagógicos, faltando ainda explorar as possíveis vantagens e adaptações necessárias para esta finalidade. A principal adaptação que deve ser feita é na forma de comunicar as hipóteses de erros ao aluno.

1.3 Padrões Elementares de Programação

Para se fazer uma proposta de um sistema de diagnóstico com fins pedagógicos é preciso antes compreender as principais dificuldades que um aprendiz de programação deve enfrentar. Pesquisas em teorias cognitivas [JS84] sobre o aprendizado de programação sugerem que programadores experientes resolvem problemas procurando soluções anteriores que estejam relacionadas com o novo problema e que possam ser adaptadas à situação atual. Por outro lado, um aprendiz que não tem nenhuma experiência anterior de programação em mente, só pode recorrer às sentenças da linguagem [Win96]. Incentivados por essas idéias, educadores de programação desenvolveram um método baseado em padrões para ensino de programação. Neste modelo, o aprendizado pode ser visto como um processo de reconhecimento de padrões, chamados de Padrões Elementares, que compara soluções gerais aprendidas (ou documentadas por educadores experientes) com a situação atual de resolução de problemas.

Padrões Elementares ajudam o aprendizado de estratégias de programação gerais recomendadas por educadores [Pro00]. Além disso, os padrões ajudam a criar uma linguagem compartilhada, entre tutor e estudante para comunicação de experiências sobre programação. Padrões Elementares podem também servir para estabelecer uma boa comunicação das hipóteses de falhas do programa do aluno geradas por um sistema automático de Diagnóstico Baseado em Modelos.

1.4 Objetivos

O principal objetivo desse trabalho é o desenvolvimento de um sistema de diagnóstico de programas, **ProPAT_deBUG** para ser usado em um IDE de Introdução à Programação. Esse sistema utiliza técnicas consagradas de Inteligência Artificial para o diagnóstico de falhas em sistemas físicos. O sistema de diagnóstico é chamado a partir do ambiente **ProPAT**, um plug-in Eclipse para programação em C usando Padrões Elementares de programação. Assim, enquanto o aluno edita um programa, ele pode acessar e inserir Padrões Elementares no programa com a intenção de satisfazer sub-metas de um dado problema. Em resumo, os principais objetivos desse trabalho são:

1. construir um sistema de diagnóstico de programas baseado em técnicas de MBD da Inteligência Artificial [MSW00] para detectar falhas no programa do aluno, chamado de ProPAT_deBUG;
2. selecionar e construir um conjunto de Padrões Elementares de programação.
3. construir um ambiente de programação com padrões, chamado ProPAT de onde o aluno poderá chamar o sistema de diagnóstico;
4. classificar falhas típicas em programas de alunos de programação;
5. avaliar o sistema de diagnóstico para um conjunto de programas com falhas selecionadas de acordo com a classificação de falhas típicas.

1.5 Organização

Essa dissertação está organizada da seguinte maneira:

Capítulo 2. Mostramos os problemas fundamentais que um aprendiz de programação deve enfrentar e uma classificação de falhas típicas em programa de alunos novatos.

Capítulo 3. Apresentamos uma visão geral das técnicas de depuração automática, dando ênfase às técnicas usadas em Tutores Inteligentes de Programação.

Capítulo 4. Apresentamos o conceito de Padrões Elementares de programação e como eles podem ser usados em um curso de Introdução à Programação, além de uma breve descrição da ferramenta ProPAT, um ambiente para aprendizes programarem usando padrões de programação.

Capítulo 5. Introduzimos os conceitos básicos de MBD e especificamos um algoritmo clássico da área de IA.

Capítulo 6. Descrevemos como construir um modelo a partir do programa do aluno e como cada uma das tarefas do processo de diagnóstico clássico foram implementadas no sistema de diagnóstico proposto.

Capítulo 7. Mostramos através de um exemplo de problema, como o sistema de diagnóstico **ProPAT_deBUG** implementado é capaz de detectar falhas em programas. Além disso, damos um exemplo de um script usado durante a comunicação da falha ao aluno.

Capítulo 8. Fazemos uma avaliação de desempenho do sistema de diagnóstico para um conjunto de programas com falhas introduzidas de acordo com a classificação de falhas.

Capítulo 9. Apresentamos as principais contribuições, trabalhos futuros e conclusões dessa pesquisa.

CAPÍTULO 2

O aprendizado de programação e uma classificação de falhas típicas

2.1 Aprendizado de programação

A Psicologia da Programação aponta dois problemas fundamentais que um aprendiz de programação deve enfrentar:

- **Aprender a linguagem de programação**: o estudante deve ser capaz de entender a sintaxe e a semântica de uma linguagem de programação.

- **Aprender a resolver problemas para o computador executar**: o estudante deve aprender a resolver problemas no computador. Isso implica no aluno ser capaz de "traduzir" uma solução conhecida (por exemplo, a resolução de uma equação de segundo grau) para uma solução que o computador execute.

Uma linguagem de programação possui muitos detalhes, mas apesar disso, não parece ser essa a maior dificuldade que o aprendiz enfrenta. O aprendiz pode saber a sintaxe e a semântica das sentenças individuais, mas não sabe como combiná-las para conseguir um programa válido [Win96]. Evidências mostram que aprender uma segunda linguagem é bem mais fácil. A hipótese é que, ao aprender uma segunda linguagem, o aluno já adquiriu a habilidade de resolver problemas de computação em diferentes linguagens de programação.

Em cursos tradicionais de programação a tarefa de construir os primeiros programas é deixada, em grande parte, ao aprendiz. O uso de Padrões Elementares (Seção 4.1.1) no ensino de programação foi proposto para ajudar o aprendiz nessa difícil tarefa.

2.1.1 Escolha do paradigma de programação para ensino

A escolha do paradigma de programação adotado numa disciplina de Introdução à Computação deve ser feita com certos cuidados. Os critérios principais para esta escolha são: as características intrínsecas do paradigma que facilitam o ensino (razões pedagógicas) e paradigmas de maior interesse pelo mercado do trabalho [DZ01].

Entre os paradigmas mais usados temos: imperativo, funcional e orientados a objetos. Foram propostos [AC01] seis modelos de implementação para a disciplina de Introdução à Computação:

- Imperativo primeiro, que usa o paradigma tradicional imperativo.
- Objetos primeiro, que enfatiza o uso de objetos e desenho orientado a objetos.
- Funcional primeiro, que introduz conceitos de algoritmos em uma linguagem com uma sintaxe funcional simples, como Scheme ou Lisp.
- Em largura primeiro, que inicia com uma visão geral da disciplina.
- Algoritmos primeiro, com foco primeiro em algoritmos descritos em pseudo-código e depois em sintaxes.
- Hardware primeiro, inicia com circuitos e camadas na hierarquia de uma máquina abstrata.

Uma das conclusões compartilhada pela maioria dos educadores é que não existe nenhum modelo ideal de implementação para uma disciplina de introdutória de programação, pois todas as abordagens têm suas vantagens e desvantagens [AC01].

O paradigma imperativo primeiro enfatiza os aspectos imperativos da linguagem como: expressões, estruturas de controle, procedimentos e funções. Neste trabalho, adotou-se o paradigma imperativo, especificamente a linguagem C. No entanto, os Padrões Elementares (Seção 4.1.1) que usaremos aqui, poderão ser facilmente adaptados para outras linguagens e paradigmas, de forma que a proposta desse trabalho independe da linguagem adotada.

2.2 Terminologia usada em Depuração

O termo falha, usado freqüentemente como sinônimo de *bug*, corresponde a um passo incorreto, processo ou definição incorreta de dados. A existência de um *bug* faz com que o programa não funcione corretamente ou não produza um resultado correto [Wie01].

Uma falha pode estar, por exemplo, no código fonte do programa ou no projeto do algoritmo. Falhas podem ter uma variedade de efeitos na funcionalidade do programa e podem não ser detectadas enquanto seus efeitos não forem percebidos ou observados.

O processo de procurar, encontrar e consertar falhas é chamado de depuração. Freqüentemente, programadores gastam muito mais esforço e tempo para encontrar e consertar falhas do que em escrever um novo código. Existem ferramentas, chamadas de *depuradores*, que ajudam, por exemplo: a monitorar a execução do programa, seguir o fluxo, verificar os valores das variáveis e parar a execução do programa em determinados pontos. Neste processo, são usados também os chamados *casos de teste* que são um conjunto de dados de entrada com os respectivos dados esperados de saída para um programa correto. No entanto, a depuração de programas continua sendo uma tarefa difícil, e quase manual.

No caso de um programador aprendiz, o professor costuma recomendar usar a simulação (também chamada de *teste de mesa*), que consiste em executar manualmente as instruções do algoritmo tomando as entradas de um caso de teste e comparando a saída do algoritmo com a saída esperada. No entanto, o processo de simulação pode não ser suficiente para o aluno descobrir e corrigir seus erros. Por exemplo, na simulação o aluno pode tentar executar o programa com suas concepções erradas ou pode identificar erros na saída, sem ser capaz de identificar as causas no programa.

Uma outra prática recomendada, e freqüentemente adotada por alunos, é verificar os valores das variáveis sem parar a execução do programa (como é feito por um depurador), mas inserindo comandos de impressão em pontos estratégicos do programa. Veremos que em um sistema de diagnóstico de programas (Cap 6) é possível que o sistema escolha automaticamente esses pontos e pergunte os valores esperados para o aluno para então compará-los com os valores gerados pelo programa.

2.3 Classificação de falhas em software

Os termos usados na depuração e teste de programas incluem [IEE90]:

- **Falha**, pode corresponder a um passo incorreto, processo ou definição incorreta de dados. O objetivo da depuração é encontrar falhas nos programas. Por exemplo, uma instrução em um programa que cause uma saída incorreta ou um erro de compilação.

- **Observação de falha**, através de dados de saída. Em diagnóstico, uma observação de falha é também chamada de *sintoma* de mal-funcionamento (Capítulo 5). Quando uma falha é introduzida num programa, é possível que ela seja percebida por meio de um resultado incorreto. Por exemplo, uma observação de falha é a discrepância entre o resultado 12 e o resultado correto que deveria ser 10.

- **Erro**, é a diferença entre um valor calculado, observado ou medido, e um valor especificado teoricamente como correto. No exemplo anterior, o erro é 2.

Alguns critérios usados para classificar falhas em software são mostrados a seguir [Wie01]:

2.3.1 Classificação de falhas pelas consequências

Falhas podem ser classificadas pela natureza das observações da falha, isto é, pelos efeitos das falhas no programa, a saber:

(C.1) **Falhas que causam consequências durante a compilação do programa**: essas falhas quase sempre podem ser localizadas de maneira eficiente por meio do uso das mensagens de erro ou avisos do próprio compilador.

(C.2) **Falhas que causam consequências durante a execução do programa**: esse tipo de falha pode ser difícil de ser detectada por um depurador, como por exemplo, um programa que termine abruptamente ou que não termine (entra em *loop*).

(C.3) **Falhas que causam consequências na saída do programa**: mesmo que o programa não termine abruptamente ou entre em *loop*, uma falha pode provocar um resultado incorreto. O erro percebido na saída pode ser usado para localizar as possíveis posições de uma falha no código fonte.

(C.4) **Falhas que não causam consequências na saída do programa**: certas falhas podem não causar erros para os casos de teste considerados. Neste caso, o processo de depuração é ainda mais difícil.

2.3.2 Classificação de tipos de falhas

Uma falha pode ainda ser classificada pela sua própria natureza, a saber:

(F.1) **Falhas sintáticas**: falhas no programa que não são consistentes com a gramática da linguagem de programação. Essas falhas são sempre detectados durante a compilação. Por exemplo, esquecer o ponto e vírgula no final da sentença em um programa escrito na linguagem C.

(F.2) **Falhas semânticas**: falhas não consistentes com a semântica da linguagem. Exemplos: declaração incorreta do tipo de dado ou uso de uma variável não inicializada. Falhas semânticas podem ser classificadas como:

(F.2.1) **Falhas semânticas causadas por uma interpretação errada do programador sobre as construções individuais da linguagem**: esse tipo de falha pode ser causado por experiências individuais próprias do programador ou conhecimento prévio. Por exemplo, interpretar uma atribuição do programa C, como o operador igualdade da aritmética; ou acreditar que em uma sentença *for*, a atualização do contador é feita inclusive na primeira vez que o laço é executado.

(F.2.2) **Falhas semânticas causadas por dificuldades no entendimento das interações e coordenação entre múltiplas estruturas**: nesse tipo de falha o programador entende cada construção individual da linguagem, mas não sabe como é a interação entre os diferentes componentes

da linguagem. Por exemplo, no caso de um programador aprendiz acreditar que em dois laços aninhados, primeiro serão executadas todas as sentenças do laço externo varias vezes e depois as instruções do laço interno.

(F.3) **Falhas lógicas**: falhas resultantes de um erro do programador no projeto do algoritmo. Por exemplo: uso incorreto de variáveis, uso de estruturas de dados erradas, etc. Essas falhas podem causar um erro de saída que pode ser percebido para a maior parte dos casos de teste. Uma vez que falhas lógicas podem ser observadas, elas constituem o principal foco da maioria dos depuradores. Tomando como base as modificações no grafo de dependências do programa [1], as falhas lógicas podem ser divididas em duas sub classes: falhas funcionais e falhas estruturais.

(F.3.1) **Falhas lógicas funcionais**: são falhas que surgem do armazenamento de um valor incorreto em alguma variável, em algum caso de execução possível (casos de teste). Falhas funcionais não alteram a estrutura do programa (o grafo de dependências do programa com falhas é equivalente ao grafo de dependência do programa correto). Em particular, essas falhas incluem o uso de um operador incorreto ou uso de literais incorretos (variáveis). Exemplos de erros funcionais são:

- omitir um operador (por exemplo, escrever i em vez de i+1);
- usar um operador incorreto (por exemplo, escrever i++ em vez de ++i) ou a variável errada (por exemplo, a[i] em vez de a[j]);
- atribuição de um valor incorreto a uma variável;
- erros em inicializações ou condições de saída que conduzem a um valor errôneo de uma variável em um laço.

(F.3.2) **Falhas lógicas estruturais**: são falhas do código fonte que alteram a estrutura do programa do aluno. Por exemplo, esquecimento de uma sentença, sentenças fora de ordem, sentenças desnecessárias ou atribuição a uma variável incorreta. Para detectar esse tipo de falha no programa do aluno é necessário compará-lo com alguma especificação, como é o caso das técnicas de verificação com respeito à especificação (Seção 3.1).

(F.4) **Falhas causadas pela interpretação do problema**: são falhas originadas pelo mal entendimento da especificação do problema. O aluno resolve um problema diferente do que foi enunciado.

[1] Estrutura para representar programas que mostra as dependências de dados e de controle para cada operação no programa [FOW87].

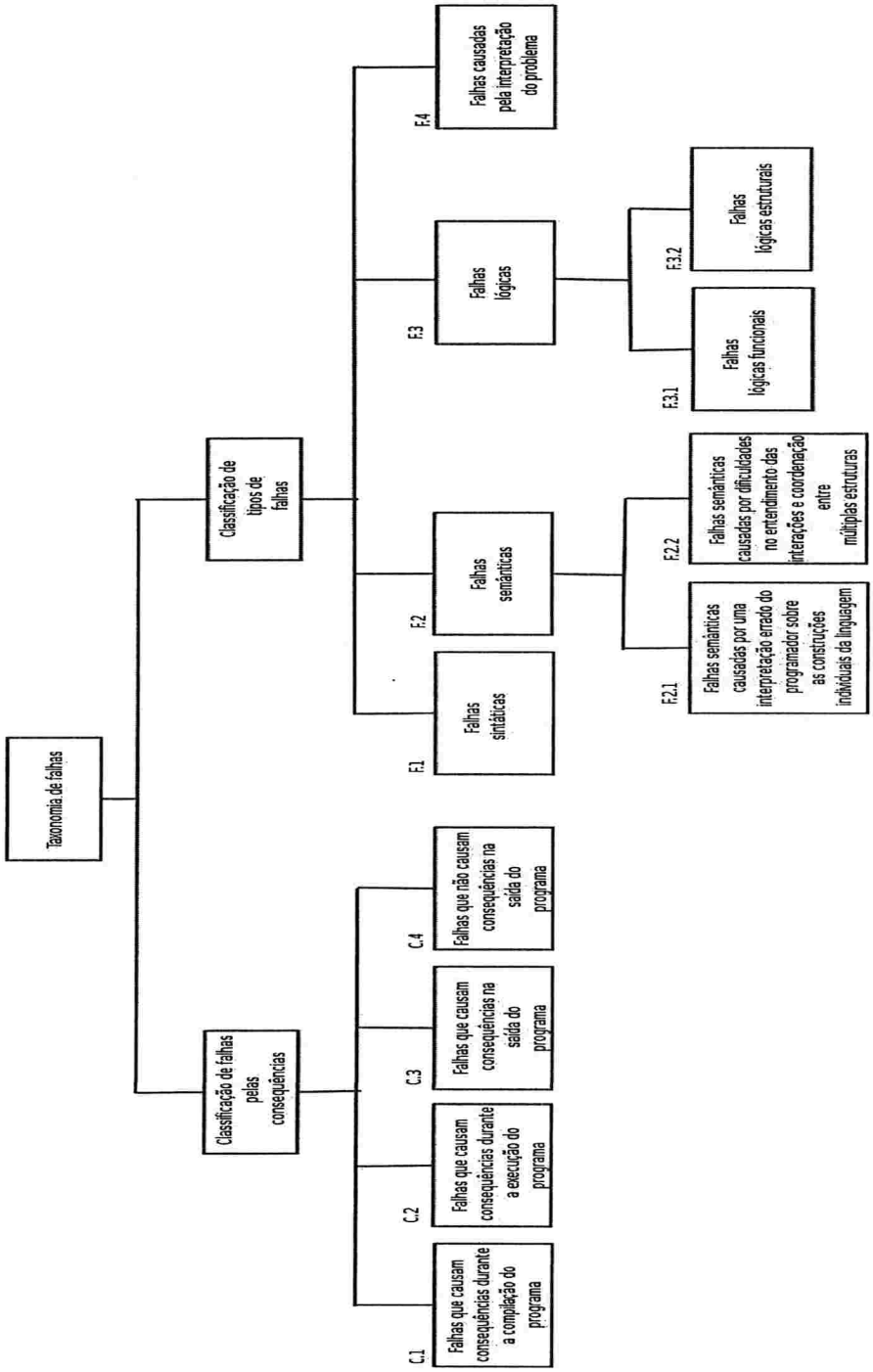


Figura 2.1: Taxonomia de falhas

O sistema de diagnóstico proposto detecta falhas que têm consequências na saída, classificadas como C.3; falhas do tipo F.3 e F.4, e algumas falhas do tipo F.2.

2.4 Falhas típicas de um aprendiz de programação

As Tabelas 2.1, 2.2 e 2.3 apresentam conjuntos de falhas típicas de alunos de uma disciplina introdutória de programação. Essas falhas foram identificadas por pesquisadores na área da Psicologia da Programação [JSGE83] [Bon85] [SS86a] [SS86b] [Sol86] [PB96] e podem ocorrer em estruturas simples (Tabela 2.1), de seleção (Tabela 2.2) e de repetição (Tabela 2.3). Cada linha das tabelas contém: o nome da falha; o nome encontrado na literatura, quando existir; uma explicação da falha; a classificação da falha pelas consequências e a classificação da falha pelo seu tipo.

O conjunto de falhas mostrado nas tabelas será usado para avaliar o sistema de diagnóstico de programas proposto nesse trabalho (Seção 8).

É importante notar que na coluna de explicação de algumas falhas é dado um exemplo, mas para outras falhas isso não é feito por motivo de espaço. Porém, exemplos para a maioria das falhas das tabelas podem ser encontrados em *http : //www.ime.usp.br/ ~ kvd/mestrado/diag_exemplos.pdf*. As falhas causadas pelo mal entendimento do problema não estão listadas nas tabelas mas exemplos de falhas desse tipo também serão usados na avaliação (Seção 8) e serão chamadas de *Interpretação1*.

Nome	Nome conhecido	Explicação da falha	Classificação pelas consequências	Classificação pelo tipo de falha
Atribuição1		O estudante interpreta uma atribuição na ordem inversa. Exemplo: a=b é interpretado como sendo a atribuição do valor de *a* para *b*.	C.1 C.3 C.4	F.1 F.2.1 F.3.2
Atribuição2		O estudante interpreta uma atribuição como uma comparação de igualdade na matemática. Exemplo: a=a+1 não é possível de fazer.	C.3	F.2.1 F.3.1
Atribuição3		A atribuição é interpretada corretamente, mas depois da atribuição a=b o estudante acredita que b não contenha mais o valor original.	C.4	F.2.1
Variável1	Nome de variável mnemônico	O estudante acredita que o significado do nome da variável, faz com que o programa selecione de maneira "inteligente" um valor baseado no significado dela. Exemplo: o aluno acredita que ao dar o nome "maior"para uma variável isso fará com que a variável automaticamente selecione o valor maior de uma lista.	C.3	F.2 F.3.2
Variável2	Mais de um valor	O estudante acredita que mais de um valor pode ser atribuído para uma variável ao mesmo tempo.	C.3	F.2 F.3.2

Paralelismo	Paralelismo	O estudante acredita que diferentes linhas seqüenciais do programa são executadas em paralelo ou que uma atribuição com variáveis define uma equação matemática que "vale" durante todo o programa. No exemplo a seguir, o aluno acredita que a área pode ser calculada nessa ordem.
		area=b*h;
		h=5;
		b=2;
			C.2 C.3	F.2.2 F.3.2

Inicialização1		O estudante acredita que as variáveis são inicializadas automaticamente com um valor quando são declaradas (somente para algumas linguagens isso é verdade).	C.1 C.3 C.4	F.2 F.3.2

Tabela 2.1: Falhas típicas em estruturas simples.

Nome	Nome conhecido	Explicação da falha	Classificação pelas consequências	Classificação pelo tipo de falha
Condicional1		Para uma sentença condicional sem ramo *else*, o aluno espera que o programa pare se a expressão condicional do *if* for falsa.	C.3	F.2.1 F.3.2
Condicional2		O aluno acredita que as sentenças dos ramos *then* e *else* sejam ambas executadas.	C.3	F.2.1 F.3.2
Condicional3		O aluno acredita que as sentenças do ramo *then* sejam executadas sempre (tanto com a expressão condicional verdadeira quanto falsa).	C.3	F.2.1 F.3.2
Condicional4		O aluno espera que a sentença que está depois do corpo de uma estrutura condicional sem *else*, seja executada como se estivesse dentro do ramo *else*.	C.3	F.2.1 F.3.2

Tabela 2.2: Falhas típicas em estruturas de seleção.

Nome	Nome conhecido	Explicação da falha	Classificação pelas consequências	Classificação pelo tipo de falha
ExpLaço1	Expressão condicional monitorada constantemente	O aluno espera que o laço termine no ponto em que sua condição falha, por exemplo no meio do laço, ao invés de esperar que o restante das instruções sejam executadas.	C.3 C.4	F.2.1 F.3.2
ExpLaço2		O aluno acredita que a variável que está na condição do *for* não tem valor dentro do laço.	C.3 C.4	F.2.1
ExpLaço3	Expressão condicional atua como restrição	O aluno interpreta um intervalo de valores na condição do *for* como uma restrição dos valores de variáveis diferentes dentro do laço.	C.3 C.4	F.2.1 F.3.2
ExpLaço4	Uma execução fora	Falha em que o laço é executado uma vez a mais ou uma vez a menos. Geralmente, esse problema acontece quando o aluno inicializa o contador do laço com zero ao invés de um, ou quando usa *menor que* ao invés de *menor ou igual que* na comparação.	C.3 C.4	F.2.1 F.3.1
Laço1		O aluno interpreta uma sentença adjacente (depois) ao laço como contida dentro do corpo do laço. O estudante não pode identificar onde inicia o laço e onde termina.	C.3 C.4	F.2.1 F.3.1
Laço2		O aluno acredita que somente a última sentença dentro do laço será executada várias vezes e as primeiras uma vez só.	C.3 C.4	F.2.1 F.3.2
Laço3		O aluno atribui um comportamento de laço para um bloco com inicio e fim.	C.3 C.4	F.2.1 F.3.2
Laço4	Dirigido pelos dados	O aluno acredita que o número de iterações do laço é controlada pelos dados, isto é, que o computador é suficientemente inteligente para determinar quando sair do laço.	C.3 C.4	F.2.1 F.3.2
Laço-Condicional1		O aluno acredita que uma variável pode ter mais que um valor e assim tratar uma sentença condicional como um laço.	C.3	F.2.1 F.3.2
Paralelismo	Paralelismo	O aluno acredita que para somar n números lidos, é preciso usar dois laços na seqüência. Um laço para fazer a leitura dos n números seguido por outro laço para fazer a soma.	C.2 C.3	F.2.2 F.3.2

Tabela 2.3: Falhas típicas em estruturas de repetição.

2.5 Considerações finais

Neste capítulo foram apontados os principais problemas que um aprendiz de programação deve enfrentar, entre eles aprender a resolver problemas para o computador executar. Também foram apresentados alguns termos usados em depuração e uma classificação de falhas pelas consequências e pelo tipo, que serão usados para avaliar o sistema de diagnóstico proposto.

O sistema de diagnóstico **ProPAT_deBUG** permite identificar falhas que causam consequências na saída do programa, mais especificamente, as falhas lógicas funcionais e estruturais, falhas causadas pela interpretação do problema e algumas falhas semânticas.

No próximo capítulo são apresentadas algumas técnicas usadas para correção automática de programas em Sistemas Tutores Inteligentes e em Sistemas de Depuração.

CAPÍTULO 3

Correção Automática de Programas em Tutores Inteligentes e em Depuradores

Na literatura de correção automática de programas existem duas categorias diferentes de sistemas: *Tutores Inteligentes de Programação* e *Sistemas de Depuração de Programas* [Duc93].

Tutores Inteligentes de Programação trabalham com programas escritos por estudantes considerados programadores aprendizes. Os programas analisados são geralmente: pequenos e escritos em uma linguagem de programação reduzida; de finalidade totalmente especificada e relacionados a problemas comuns. Portanto, suas soluções podem ser repetidas. Para esses sistemas, é feita a suposição de que o aprendizado ocorre na comunicação entre o tutor e o estudante durante o processo de depuração, sendo que a comunicação serve para identificar as intenções do aluno ao escrever seu código fonte.

Sistemas de Depuração de Programas trabalham com programas escritos por programadores experientes. Os programas são geralmente grandes, originais e complexos. Esses programas não podem ser completamente especificados no início e eles usam a maioria das instruções e atributos da linguagens de programação. O processo de depuração não tem a finalidade de ensinar, mas de encontrar falhas do programa. Assim, em tais sistemas, a comunicação ocorre somente para a aquisição de dados durante a depuração e não para fins de aprendizagem.

3.1 Visão Geral de Técnicas de Correção Automática de Programas

Em [Duc93] é apresentada uma visão geral das **técnicas automáticas** de correção de programas empregadas em: *Tutores Inteligentes de Programação* e *Sistemas de Depuração de Programas*. Nesse trabalho, são definidas três técnicas gerais: ***verificação com respeito à especificação***, ***verificação com respeito ao conhecimento da linguagem*** e ***filtragem com respeito aos sintomas***. Muitos sistemas combinam estas técnicas de acordo com o conhecimento que o sistema tem acesso.

A técnica de **verificação com respeito à especificação** compara o programa real com uma especificação formal existente (supondo que a intenção do programador foi a de implementar essa especificação). Na técnica de **verificação com respeito ao conhecimento da linguagem**, o conhecimento é geralmente com relação às restrições da linguagem (por exemplo, em C, variáveis devem ser declaradas antes de serem usadas). Esse tipo de depuração procura um pedaço de código que não obedece algum conhecimento da linguagem de programação. As técnicas de verificação com respeito à especificação e verificação com respeito ao conhecimento da linguagem são também usadas em métodos formais de verificação de programas porém, por serem muito complexas, não existem depuradores automáticos que as implementem. Por outro lado, como essas técnicas podem ser aplicadas somente a programas pequenos, elas tem sido exploradas pelos sistemas Tutores Inteligentes de Programação.

Nos *Sistemas de Depuração de Programas*, ***filtragem*** é a técnica mais usada. A técnica de filtragem se concentra nos sintomas anormais (erros de saída) para encontrar o erro do programa. A estratégia de **filtragem** reduz o espaço de busca ao presumir corretas as partes do programa que não podem ter provocado o sintoma observado de falha. Com esta estratégia, no pior caso, o programa inteiro será analisado, ao contrário da estratégia de verificação com respeito ao conhecimento da linguagem, em que podem ser geradas mais hipóteses de erros do que o número de linhas do código fonte. Assim, uma técnica de filtragem deveria ser usada como uma etapa preliminar na presença de um sintoma de erro, antes de se aplicar os dois métodos anteriores.

A Figura 3.1 mostra as técnicas mais usadas em Sistemas Tutores Inteligentes e em Sistemas de Depuração de Programas. Essa figura apresenta parte da classificação proposta por [Duc93] e [Wie01], enfatizando exemplos de técnicas de filtragem, a saber:

- *Depuração Algorítmica* (*algorithmic debugging*), também conhecida como depuração declarativa, se baseia em encontrar falhas no nível de chamadas a métodos. Essa técnica necessita de bastante interação com o usuário.
- *Corte do Programa* (*program slicing*), poda todas as sentenças que não causam o valor errado ou a seqüência de controle errada. O programa restante, chamado de corte ou fatia, deveria conter a falha. Porém, isso não é garantido, o que torna essa técnica incompleta.
- A técnica de *Mutação do Programa* (*program mutation*), modifica partes pequenas do programa e se

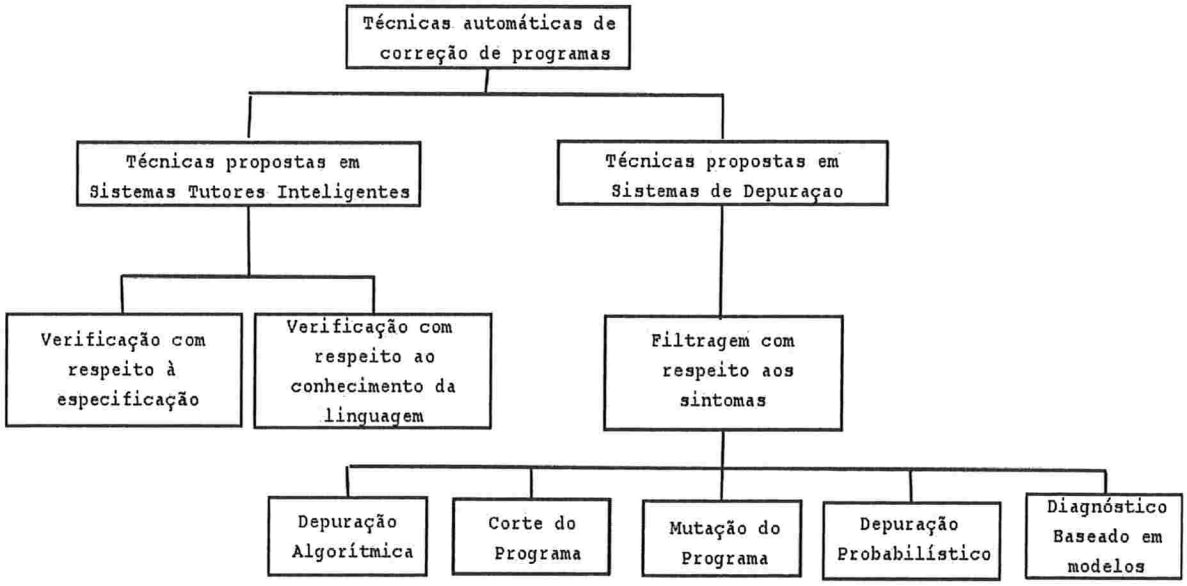


Figura 3.1: Taxonomia de estratégias de correção automática

o programa "mutante" não elimina os sintomas de falha, as peças que foram modificadas são supostas corretas. Os mutantes não só fornecem a posição do erro, como também podem dar sugestões de reparo.

- *Depuração Probabilística* (*probabilistic debugging*), usa probabilidades de falhas especificadas por especialistas. Computa todas as posições de falha potenciais e usa uma rede de crenças para encontrar as sentenças defeituosas mais prováveis .

- *Diagnóstico Baseado em Modelos* (*model base diagnosis - MBD*), é um tipo de técnica de filtragem que obteve muito sucesso no diagnóstico de sistemas físicos e que também tem sido usada para diagnóstico de programas para auxiliar programadores experientes na depuração de seus programas [SW98]. Essa técnica corresponde ao estado da arte das técnicas de IA para diagnóstico automático. Os primeiros trabalhos se baseavam na programação lógica, linguagens de projeto de hardware como VHDL (VHSIC Hardware Description Language) e linguagens funcionais. Trabalhos recentes usam linguagens imperativas com execução seqüencial e linguagens orientadas a objetos como Java ou C++. MBD usa uma descrição lógica do sistema de software, composta por componentes e conexões (modelo estrutural) e uma descrição desses componentes em termos de entradas e saídas esperadas (modelo comportamental). A descrição é baseada no código do programa e na informação sobre a semântica da linguagem de programação. O aspecto central para o sucesso desta estratégia é a construção de modelos eficazes.

3.2 Análise das Técnicas de Correção Automática de Programas

Todas as técnicas até então propostas para *Tutores Inteligentes de Programação* e *Sistemas de Depuração de Programas* são consideradas insuficientes para tratar casos reais.

As técnicas geralmente usadas por *Tutores de Programação* (verificação com respeito à especificação e verificação com respeito ao conhecimento da linguagem) que podem encontrar erros com precisão, são muito caras em termos computacionais para serem usadas em todo um programa real e complexo.

Por outro lado, as técnicas geralmente usadas em *Sistemas de Depuração de Programas* (técnicas de filtragem) são consideradas práticas mas encontram erros aproximados. Além disso, métodos que procuram por erros, meramente inspecionando o código, não podem lidar com uma grande variedade de problemas de lógica de programação, uma vez que eles não são capazes de reconhecer que erros não-sintáticos não são uma propriedade intrínseca dos programas defeituosos mas sim residem na relação entre a intenção do programador e sua realização no código [Wen87].

As técnicas de **filtragem** apesar de serem mais eficientes do que outros métodos, ainda não está claro como esse processo pode ajudar os estudantes a aprenderem a programar, sendo que a maior dificuldade reside na construção de um diálogo, baseado em um raciocínio lógico, que promove o aprendizado.

3.3 Tutores Inteligentes de Programação

Um Sistema Tutor Inteligente (ITS) é um sistema de ensino/aprendizagem que emprega técnicas de IA com o objetivo de promover o aprendizado individualizado. Para isso, um ITS deve ser capaz de diagnosticar as soluções do aluno, construir um modelo do seu conhecimento (*diagnóstico cognitivo*), fazer o planejamento curricular e planejar sua comunicação com o aluno [Wen87]. A comunicação com o aluno pode também se dar durante o diagnóstico de soluções do aluno.

Como falamos anteriormente, uma das técnicas comumente encontradas em Sistemas Tutores Inteligentes de Programação, é a de verificação com respeito à especificação que compara o programa real com uma especificação formal existente supondo que a intenção do programador foi a de implementar essa especificação. Uma abordagem mais simples para diagnosticar o programa do aluno a fim de achar erros não-sintáticos é procurar por anomalias na estrutura e comportamento do programa. No entanto, essa abordagem não é capaz de ajudar um aprendiz a identificar suas "intenções" num código sem anomalias óbvias (laços infinitos ou erros típicos de principiantes). Não existe ainda uma proposta que combine esses dois aspectos (sendo esse um dos objetivos desse trabalho, discutido na Seção 4.3.3).

Na Seção 3.3.1 faremos uma revisão dos principais ITSs cuja proposta foi a de identificar as intenções do aluno, refletidas no programa.

3.3.1 Exemplos de Tutores Inteligentes de Programação

A maioria dos sistemas tutores de programação desenvolvidos até os dias de hoje partiram do princípio que é preciso identificar as intenções do aluno na construção do programa, seja através da comunicação com o aluno ou através de processos automáticos para reconhecimento de planos [Joh90]. Isso porque, um sistema de depuração automática que procura por anomalias óbvias no programa, não é capaz de apontar erros de alto nível do aluno, isto é, apontar as causas dos erros do programa que ao serem comunicados promovam o aprendizado.

Os sistemas tutores LAURA [AJ80] e TALUS [Mur89], desenvolvidos nos anos oitenta, podem ser considerados os primeiros sistemas tutores para programação baseados em intenção. Ambos analisam os programas comparando-os com soluções ideais fornecidas pelo professor, consideradas como intenções de programação do aluno. A grande limitação destes sistemas é que eles dependem de soluções ideais fornecidas por um professor e que nem sempre correspondem de fato às intenções dos alunos.

O sistema TALUS simplifica o programa do estudante feito em LISP tal que a semântica seja única. Em seguida, o programa é representado por uma árvore que é comparada com um conjunto de funções implementadas para reconhecer o algoritmo usado [SW98]. Isto torna o sistema TALUS um pouco mais geral que o sistema LAURA, que somente faz a comparação com uma única solução fornecida pelo professor.

O sistema LISP TUTOR [AS86] trabalha com regras de produção para gerar o conjunto dos próximos passos da resolução de um problema. O conjunto de regras representa um conjunto de possíveis combinações de soluções corretas. Se o estudante faz uma movimentação diferente dos passos preditos, o tutor dá alguma realimentação (*feedback*) ao estudante, isto é, tenta recolocar o estudante num conjunto esperado de caminhos que levam à solução. Uma das vantagens do LISP TUTOR é que ele fornece ao usuário uma resposta imediata quando erros são detectados. A desvantagem é que, como nos dois sistemas descritos acima, o sistema LISP TUTOR restringe a liberdade do estudante na implementação do programa.

O KUMAR'S TUTOR [Kum02] é um tutor baseado em modelos que explica, linha a linha, o programa do aluno para ajudá-lo a entender o comportamento do código e predizer as saídas de programas em C++, identificando falhas semânticas e falhas que causam conseqüências durante a execução do programa. A capacidade de simular o comportamento do programa é o que torna esse sistema interessante porém, o custo de desenvolver um simulador desse tipo para cada linguagem de programação pode ser muito alto.

O Sistema Tutor PROUST [JS84] foi desenvolvido com base na teoria da psicologia de programação e representa um marco importante na evolução dos sistemas tutores de programação, além de ser o trabalho mais citado da área. PROUST [Joh90] aprimorou a idéia de identificação das intenções do aluno empregadas pelos sistemas TALUS, LAURA e LISP TUTOR. Isto é feito através do casamento entre sub-metas do problema e planos de programação identificados no programa do aluno.

PROUST começa a analisar o programa do aluno a partir de uma *especificação declarativa do problema*,

isto é, uma agenda de metas e sub-metas a serem satisfeitas, especificadas pelo professor. Johnson [Joh90] propõe um método para determinar os erros no programa que o aluno está tentando construir e as idéias erradas que ele poderia ter para explicar a presença de erros, que envolve dois passos:

1. identificação dos fragmentos funcionais no programa que foram usadas para atingir as metas (implementação dos planos de programação do aluno) e
2. reconstrução das metas que o estudante está tentando satisfazer.

Para compreender e fazer diagnóstico do programa do estudante, PROUST utiliza:

- o conjunto de objetivos do problema (dado pelo professor);
- a base de conhecimento (biblioteca) de planos de programação (dados pelo professor), usadas para identificar os planos de programação no código do aluno e
- uma biblioteca de erros comuns (associados aos planos da biblioteca (dados pelo professor).

A tarefa de reconhecimento dos planos de programação do aluno no código é a base para o entendimento do mesmo. No entanto, uma vez que programadores aprendizes podem construir programas de formas não previstas pelo professor, PROUST pode falhar em sua tarefa de reconhecimento de planos.

PROUST possui uma hierarquia de metas de programação que é usada para decompor objetivos de problemas de forma hierárquica. Através dos planos identificados, os objetivos são decompostos em sub-objetivos ou em novos planos (pedaços de programas) que não podem mais ser decompostos. Essa decomposição pode não ser única e os programas implementados pelo aluno podem ser associados a mais de uma decomposição. Os erros (código cujo comportamento não concorda com a especificação do programa) são descobertos com a comparação de planos da biblioteca com o programa do aluno. Programas com erros podem ser derivados de uma decomposição incorreta de objetivos ou de implementações incorretas de decomposições corretas de objetivos, por parte do aluno. PROUST começa selecionando um objetivo da descrição do problema, recuperando o conjunto de planos da base de conhecimento que satisfazem (implementam) o mesmo, para então compará-los com o código. Uma vez que planos podem ter sub-objetivos, este processo é recursivo.

O exemplo de problema dado a seguir [JS84], ilustra de forma clara como PROUST funciona.

Problema 1: *Ler números calculando sua soma até que o número 99999 seja lido. Calcular a média. Não inclua o número 99999 no cálculo da média.*

A Figura 3.2 mostra a decomposição de objetivos em sub-objetivos e planos para o Problema 1, e a Figura 3.3 mostra como essa decomposição está relacionada ao código de uma solução correta desenvolvido em C. Os objetivos principais do Problema 1 são: calcular a média e mostrar a média. Para calcular a média o programador tem que calcular a soma, calcular o contador e calcular a divisão. O primeiro sub-objetivo,

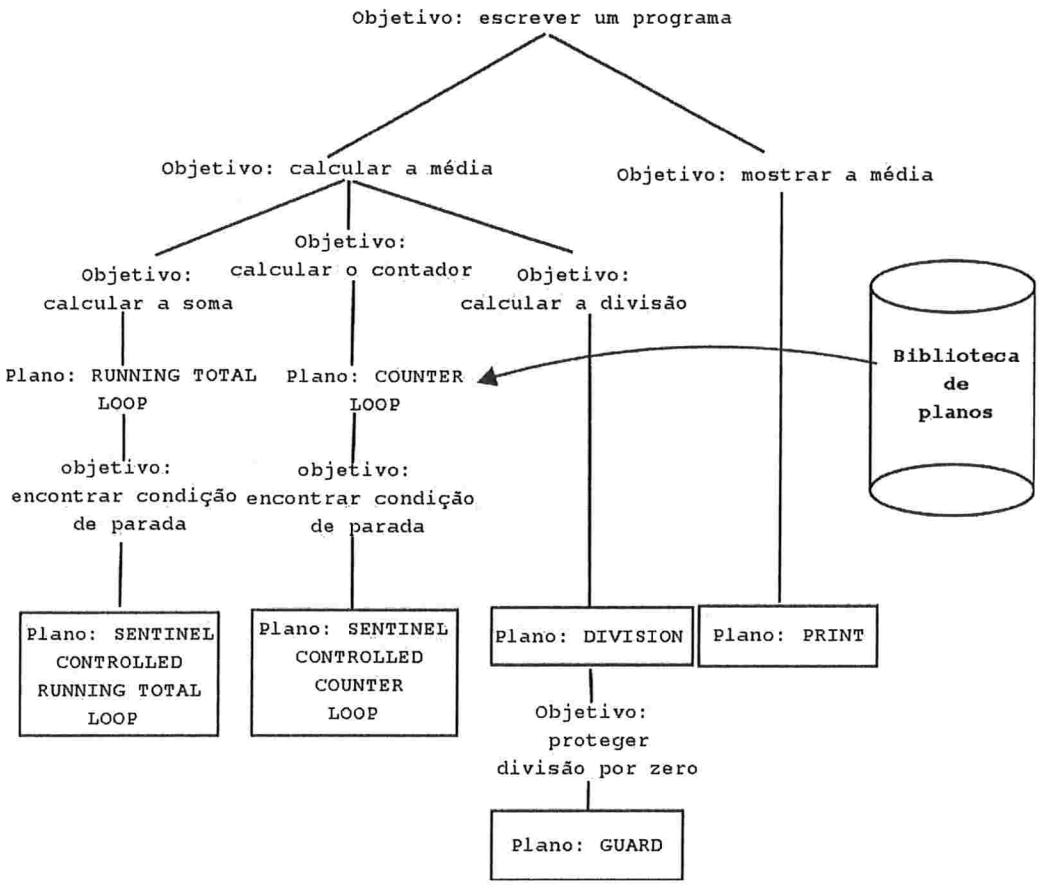


Figura 3.2: Decomposição de objetivos em sub-objetivos e planos para o problema 1

calcular a soma, pode ser atingido usando o plano: "Running Total Loop", que por sua vez inclui o sub-objetivo encontrar a condição de parada e está relacionado com o plano "Sentinel Controlled Running Total Loop". Esse plano pode ser identificado nas linhas 4,6,7 e 9 na Figura 3.3.

Com base no sistema PROUST, Lane and VanLehn [LV03] [LV04] desenvolveram o tutor PROPL que usa a interação em linguagem natural para ajudar os alunos a entenderem e planejarem seus programas antes de começarem a escrever o programa em uma linguagem de programação. Durante a sessão no tutor PROPL, o aluno deve:

- identificar objetivos do problema de programação;

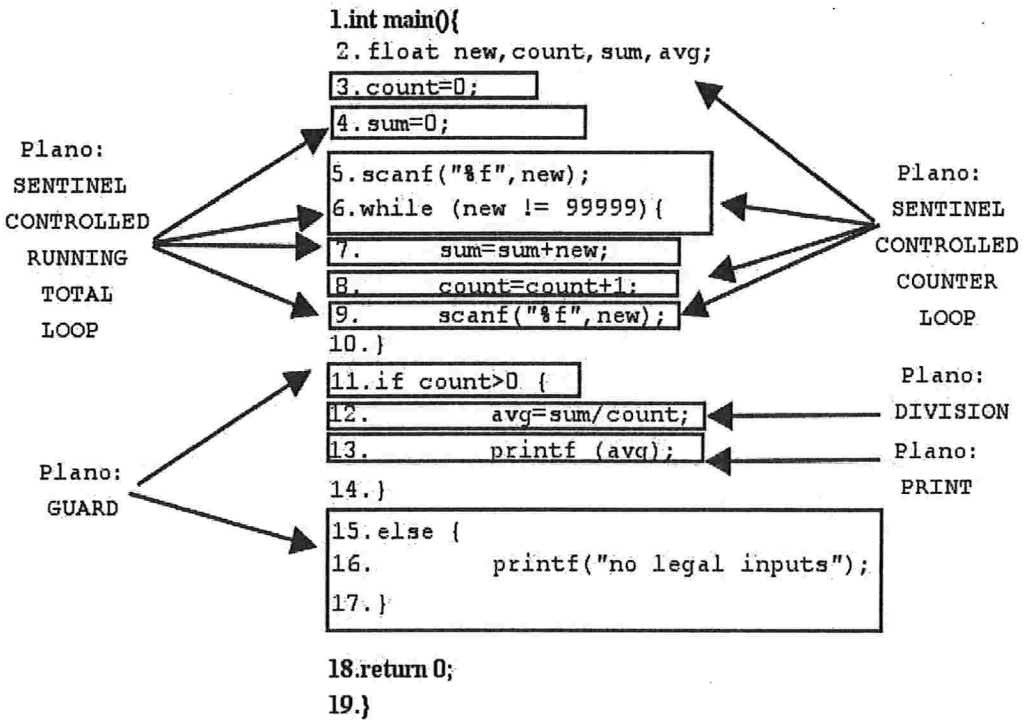


Figura 3.3: Planos usados para o problema 1

- especular as possíveis maneiras alternativas de atingir os objetivos (por meio de um diálogo);
- descrever um plano para satisfazer os objetivos do problema, sugerindo passos, descritos em pseudo-código e
- compor o programa completo em pseudo-código.

3.3.2 Comparação entre Sistemas Tutores Inteligentes e Sistemas Gerenciadores de Aprendizagem

Sistemas Gerenciadores de Aprendizagem (LMS-*Learning Management System*) são sistemas integrados que dão suporte a vários professores e estudantes. LMSs oferecem ao professor a possibilidade de compor suas disciplinas a partir de unidades de aprendizagem novas ou já existentes (chamadas LO-*Learning Objects*). Esses objetos de aprendizagem são modelados e descritos com base em estruturas padrões e metadados. Isso significa que um LO poderia ser reusado em muitas disciplinas e para diferentes propósitos [SGD04].

Enquanto os ITSs estão preocupados com a adaptação ao estudante, o LMS está principalmente focado na reusabilidade dos LOs e na execução de tarefas administrativas e de colaboração.

3.4 Considerações finais

Nesse capítulo foram apresentadas as técnicas usadas em correção automática de programas, verificação com respeito à especificação, verificação com respeito ao conhecimento da linguagem e filtragem. As duas primeiras têm sido usadas por Sistemas Tutores Inteligentes e a terceira tem sido usada por Sistemas de Depuração. Também foram apresentados alguns Tutores Inteligentes de Programação que tentam identificar as intenções do aluno na construção do programa.

O sistema de diagnóstico **ProPAT_deBUG** combina duas técnicas: verificação com respeito ao conhecimento da linguagem e filtragem com respeito aos sintomas, como será mostrado em mais detalhes no Capítulo 4, na Seção 4.3.3.

No próximo capítulo é apresentado o projeto ProPAT um ambiente para cursos de introdução à programação em que o estudante pode escolher exercícios e construir programas usando *Padrões Elementares* de programação (soluções recomendadas por educadores para problemas recorrentes).

CAPÍTULO 4

O projeto ProPAT

Como vimos, a maioria dos sistemas tutores de programação adotam a técnica de verificação de programas com respeito à especificação para a correção automática do programa do aluno. Um dos problemas dessa abordagem é ter que prever todas as possíveis soluções (programas) para um dado problema. Por exemplo, no sistema tutor PROUST, ainda que os planos representados internamente sejam aqueles mais provavelmente usados por um aluno de programação, PROUST falha para soluções não previstas.

Nesse capítulo, introduzimos o conceito de Padrões Elementares de programação e mostramos como eles podem ser vistos como planos de programação. Apresentaremos também uma breve descrição da ferramenta PROPAT, um ambiente para aprendizes programarem usando Padrões Elementares.

Diferente de PROUST, PROPAT faz a correção automática do programa do aluno usando técnicas de diagnóstico baseado em modelos (como veremos nos Capítulos 5 e 6), e usa os Padrões Elementares como base para a construção de diálogos usados para comunicar e discriminar as hipóteses de falha no programa do aluno geradas pelo diagnóstico, como será visto no exemplo do Capítulo 7.

4.1 Ensino de programação baseado em Padrões Elementares

Com base nas idéias da psicologia de programação (Seção 2.1) e do sistema PROUST [Joh90], a comunidade de Padrões Pedagógicos desenvolveu um modelo baseado em padrões para ensino de programação [ETW+96]. Neste modelo, o aprendizado pode ser visto como um processo de reconhecimento de padrões que compara experiências passadas, ou soluções recomendadas por educadores de programação, com a situação atual de resolução de problemas.

A abordagem para ensinar programação é apresentar aos estudantes pedaços pequenos de programas ao invés de esperar que eles escrevam programas inteiros a partir do zero (de um programa vazio). Esses pedaços de código são chamados **Padrões Elementares** de programação, descritos em termos de uma situação

de programação, com exemplos de aplicação em uma linguagem de programação, como Java, C ou C++. Os padrões podem ser ensinados pelo professor em sala de aula ou documentados na forma impressa ou digitalizada. Padrões são projetados por um professor experiente ou recomendados por um grupo de educadores de programação. Padrões têm a intenção de satisfazer sub-metas de problemas de programação sendo necessário que o aluno complete, adapte ou instancie o padrão utilizado para obter um programa completo e compilável.

Supondo que os estudantes usarão os Padrões Elementares aprendidos para construírem seus programas, um Sistema Tutor Inteligente teria um número de vantagens a partir desta estratégia de ensino, a saber:

- identificar as intenções de programação do aluno com base nos Padrões Elementares usados;
- servir como material de estudo para o estudante, uma vez que a documentação de um padrão incluída no sistema define os termos e conceitos que o estudante precisa aprender;
- estabelecer os termos e conceitos definidos nos padrões como linguagem para comunicação entre sistema e aluno.

4.1.1 Padrões Elementares

O objetivo dos padrões dentro da comunidade de software é criar um corpo de literatura que ajude aos desenvolvedores de software a resolverem problemas recorrentes encontrados no processo de desenvolvimento. Padrões ajudam a criar uma linguagem para comunicar experiências sobre problemas e suas soluções. Representando formalmente essas soluções e suas relações, podemos capturar o conhecimento que corresponde a um entendimento de senso comum sobre boas soluções e arquiteturas. Ter uma linguagem padrão para comunicar as estruturas e estratégias de soluções permite o raciocínio claro sobre elas. O foco principal não está na tecnologia, mas na criação de uma cultura para documentar e dar suporte na engenharia de projetos e arquitetura [App00].

Similares aos Padrões de Programação, os *Padrões Elementares* de programação descrevem problemas comuns a programadores novatos. Esses problemas e suas soluções são descritas em um formato que, entre outras coisas, facilite o reuso. Padrões Elementares devem ser simples, concisos e seu uso no contexto de aprendizes é recomendado por pesquisadores em ensino de programação [Pro00].

Em geral, um padrão associa um problema a uma solução e fornece informação sobre a situação em que ele pode ser aplicado. Seu uso potencial no contexto de aprendizagem de programação tem sido explorado pela comunidade de Padrões Pedagógicos [Pat] [Gro]. Padrões Elementares estão disponíveis na Web para as linguagens C, C++ e Java [Wal01]. Ao longo da década de 90, professores de programação têm documentado padrões que todo aprendiz de programação deve saber, incluindo: *padrões de seleção* [Ber99], *padrões de repetição* [AW98] e outros [Bri02]. Porter e Calder [PC03] sugerem um processo para aplicar padrões de programação no ensino em sala de aula e Proulx [Pro00] criou uma estrutura para um primeiro curso em Ciência da Computação baseado em Padrões Elementares de programação.

4.1.2 Vantagens dos Padrões Elementares no processo de aprendizagem

Fazendo a suposição que Padrões Elementares de programação podem ser compreendidos e utilizados por alunos de programação, eles podem ajudar o aprendizado:

1. de estratégias gerais (de mais alto nível de abstração) e
2. da linguagem de programação, uma vez que sua documentação contém um programa de exemplo da aplicação do padrão.

Além disso, supondo que eles possam ser identificados no programa do aluno pelo professor, eles também ajudariam o aprendizado, nos seguintes aspectos:

1. para que o professor identifique as intenções do aluno e
2. para estabelecer uma melhor comunicação entre professor e aluno, já que eles fornecem um vocabulário sobre estratégias gerais de solução de problemas que ambos têm acesso.

Durante a programação, o aluno pode modificar ou intercalar padrões ficando difícil para o professor identificá-los na versão final do programa do aluno. Neste caso, assistir a inserção de padrões pelo aluno durante a programação pode ser a melhor maneira de saber que padrões foram usados. Essa é a proposta do ambiente de programação ProPAT (Seção 4.3).

4.2 Documentação e Exemplos de Padrões Elementares

A Tabela 4.1 descreve a estrutura geral de *Design Patterns* [GHJV95], que têm sido muito usados para o desenvolvimento de projetos de programação orientada a objetos. Com base nessa estrutura foram criados os Padrões Elementares. Na Tabela 4.2 é mostrado como os campos propostos para documentar um *Design Pattern* foram adaptados para Padrões Elementares. A tabela mostra quais campos de um documento de Padrões Elementares mantêm o mesmo uso adotado em *Design Patterns*. Para os campos modificados ou adaptados, damos algumas explicações a seguir.

Nos Padrões Elementares, a **Intenção** foi transformada em **Intenção Pedagógica** e **Intenção do Padrão**. A intenção pedagógica pode ser usada para descrever o objetivo pedagógico que um padrão pretende alcançar. A **Intenção do Padrão** é a mesma definida para *Design Patterns*. A **Intenção do Padrão** pode ser vista como a intenção do aluno

É importante observar que, em sistemas tutores, como por exemplo o PROUST, a intenção do aluno corresponde ao *plano de programação* empregado por ele. PROUST tenta identificar esses planos no programa

do aluno. Fazendo-se um paralelo com o uso de Padrões Elementares, a intenção do aluno pode ser diretamente relacionada a Aplicabilidade e Estrutura da Solução do padrão elementar empregado pelo aluno.

Acreditamos que o campo **Motivação** é muito importante na proposta de usar Padrões Elementares como material pedagógico, tanto para o professor como para o aluno. No entanto, a maioria dos Padrões Elementares encontrados na literatura não preenchem esse campo. Nesse trabalho, usamos a Motivação para contextualizar os problemas, e por conseqüência os padrões, em uma área de aplicação específica[1]. Por exemplo, para ensinar Introdução à Computação para estudantes de engenharia, seria mais adequado propor problemas e padrões relacionados à área de controle de robôs.

A **Estrutura da Solução** foi dividida em **Sintaxe** e **Semântica**. Na **Sintaxe** é descrito *o que* o padrão faz através de um pseudo-código da solução que inclui comandos da linguagem selecionada e metadados (dados sobre dados), usados para descrever termos e conceitos gerais de programação. Essa foi uma decisão adotada nesse projeto e que foi encontrada em apenas um trabalho da literatura [Bri02]. Acreditamos que metadados servirão para a construção de uma ontologia sobre introdução à programação [SGD04], com grande utilidade para tutores inteligentes e LMS (*Learning Management System*) de programação (ver Seção 9.1). Na **Semântica** é descrito *como* o padrão realmente funciona através de um texto e de uma estrutura de controle.

O campo **Aplicabilidade**, é o mesmo que em *Design Patterns*. Para que o aluno seja capaz de compreender claramente em que situação o padrão se aplica, recomendamos fortemente que sua descrição seja relacionada à Motivação. Por exemplo, no caso da motivação ser a área de controle de robôs, o padrão *repetição contada* da Tabela 4.3 traria em **Aplicabilidade** o seguinte texto: " *Você quer repetir um determinado número de vezes um conjunto de ações. Em geral o conjunto de ações está relacionado à movimentação do robô por uma linha ou coluna de uma matriz de posições. A posição inicial deve ser lida e as demais devem ser geradas de acordo com a trajetória do robô*".

O campo **Participantes e Colaboradores** de *Design Patterns*, será chamado de **Pré-requisitos**. Esse campo, ao invés de definir as classes que participam de um padrão, e as colaborações entre elas, definirá os conceitos e definições de programação que o aluno deve saber para aprender o Padrão Elementar. Esse campo está fortemente relacionado ao conjunto de metadados declarados na Estrutura da Solução.

O campo **Implementação** deve dar dicas e sugestões ao aluno sobre aspectos relevantes na implementação do padrão que geralmente representam dificuldades para um programador principiante,.

O campo **Usos Conhecidos**, ao invés de apontar exemplos do padrão encontrado em sistemas reais, deve dar exemplos de problemas, eventualmente já resolvidos pelo aluno, em que o padrão foi usado.

[1]Na Universidade de São Paulo, a disciplina de Introdução à Computação é ministrada para vários cursos de áreas diferentes: Ciências Exatas e Tecnologia, Ciências Humanas e Biológicas, Ciências da Natureza e para Engenharia. Muitos educadores acreditam que o domínio de aplicação (os problemas de programação propostos e a linguagem de programação adotada) deveriam ser diferentes para cada uma dessas áreas, ou seja, contextualizados à área de atuação do aluno, como um fator de motivação.

Alguns exemplos de Padrões Elementares são mostrados na Tabela 4.3, contendo partes da documentação de três padrões de laço: *Repetição com sentinela*, *Repetição contada* e *Repetição com indicador de passagem*. Note que eles correspondem a estratégias elementares distintas comumente usadas para resolver problemas de processamento de seqüências, como é mostrado na coluna *Aplicabilidade* da Tabela 4.3. Um estudante que use uma estratégia quando outra é mais naturalmente aplicável, terá dificuldades em completar sua tarefa de programação. Na Figura 4.4 é mostrada a *Motivação* para os três padrões da Tabela 4.3 e nas Figuras 4.1 e 4.2 o padrão *Repetição com Sentinela* é descrito em maior detalhes.

DESIGN PATTERN	
Elemento	Descrição
Nome do padrão e classificação	o nome do padrão indica a essência do padrão (problema, solução e conseqüências). Um bom nome é imprescindível, já que será parte do vocabulário do projeto de programação.
Intenção (Intent)	uma sentença curta que responde às seguintes questões: o que o padrão faz; qual é a sua intenção; e para que problema ele está direcionado.
Nomes alternativos	outros nomes conhecidos do padrão.
Motivação	um cenário que ilustra o problema de programação e como este é resolvido pelo padrão. O cenário ajuda a entender melhor a descrição abstrata do padrão.
Estrutura da Solução	uma representação gráfica do padrão (como um diagrama de classes). A estrutura descreve os elementos usados na solução do padrão. Uma recomendação é que a solução não deve descrever um projeto completo dado que o padrão é como um molde que pode ser reusado em diferentes situações, isto é, deve ser uma descrição abstrata de como resolver o problema. Assim, é função do usuário completá-lo, adaptá-lo ou instanciá-lo para o problema corrente.
Aplicabilidade	quais são as situações em que o padrão pode ser aplicado, incluindo exemplos de projetos ruins para os quais o emprego do padrão poderia melhorar um programa.
Participantes e colaborações	classes e objetos, suas responsabilidades e colaborações.
Conseqüências	como o padrão atinge seus objetivos; quais são os os resultados e compromissos (*trade-offs*) de se aplicar o padrão; ou ainda podem ser críticas para a avaliação de alternativas ilustrando custos e benefícios do uso do padrão.
Implementação	que sugestões ou técnicas deveriam ser conhecidas quando se implementa o padrão, incluindo aspectos específicos da linguagem.
Código exemplo	fragmentos de código fonte que ilustram como se deve implementar o padrão em uma linguagem de programação específica.
Usos conhecidos	exemplos do padrão encontrados em sistemas reais. Deve-se incluir pelo menos dois exemplos em domínios diferentes.
Padrões relacionados	quais padrões estão diretamente relacionados; quais as principais diferenças; com que outros padrões este padrão deveria ser usado.

Tabela 4.1: Estrutura de um documento de ***Design Patterns***

PADRÃO ELEMENTAR	
Elemento	Descrição
Nome do padrão e classificação	nomes de Padrões Elementares são simples sendo que alguns deles são nomeados pelas próprias estruturas de controle básicas.
Intenção do padrão	o mesmo que em *Design Patterns*
Intenção Pedagógica	a intenção pedagógica do padrão é definida pelo professor.
Nomes alternativos	o mesmo que em *Design Patterns*
Motivação	o mesmo que em *Design Patterns*
Estrutura da Solução 1: Sintaxe	a sintaxe do padrão é um pseudo-código que inclui: código, e *metadados* que são usados para descrever termos e conceitos gerais de programação.
Estrutura da Solução 2: Semântica	a semântica descreve como o padrão realmente trabalha (em um formato textual) e o fluxo de controle (em um diagrama).
Aplicabilidade/Objetivo	o mesmo que em *Design Patterns*, porém, para fins pedagógicos, sua descrição deve ser fortemente baseada no que foi descrito para o campo Motivação
Pré-requisitos	conceitos e definições que o aluno já deverá saber para aprender o padrão e que podem estar relacionados ao conjunto de metadados.
Conseqüências	o mesmo que em *Design Patterns*
Implementação	sugestões que deveriam ser conhecidas pelo aluno para implementar o padrão. Por exemplo, definição de tipos de dados, evitar a divisão por zero, entre outras.
Código exemplo	o mesmo que em *Design Patterns*
Usos conhecidos	poderia ser mostrados exemplos de uso do padrão encontrados em exercícios já resolvidos pelo aluno
Padrões relacionados	o mesmo que em *Design Patterns*

Tabela 4.2: Estrutura de um documento de *Padrões Elementares*

Nome do padrão	Aplicabilidade/objetivos	Estrutura da Solução 1: Sintaxe

Repetição contada	Você quer repetir um determinado número de vezes um conjunto de ações. Os elementos podem ser lidos ou gerados. Em geral, o conjunto de ações está relacionado ao processamento de uma seqüência de elementos. A quantidade de elementos deve ser conhecida e fazer parte da condição da repetição.
	Além disso, uma variável chamada contador deve ser usada para contar os elementos já processados.
		<INICIALIZAÇÕES>
		<INICIALIZAÇÃO DO CONTADOR>
		while (<CONDIÇÃO DO CONTADOR>){
		  <LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQÜÊNCIA>
		  <PROCESSAR ELEMENTO>
		  <ATUALIZAÇÃO DO CONTADOR>
		}

Repetição com sentinela	Você quer repetir um conjunto de ações, que em geral, está relacionado ao processamento de uma seqüência de elementos. Os elementos da seqüência podem ser lidos ou gerados. A quantidade de elementos é desconhecida mas o fim da seqüência é indicado por um valor sentinela. Isso quer dizer que, o número de repetições depende da variável sentinela, ou seja essa variável deve estar na condição da repetição	<INICIALIZAÇÕES>
		<INICIALIZAÇÃO DA VARIÁVEL SENTINELA>
		while (<CONDIÇÃO DA VARIÁVEL SENTINELA>){
		  <LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQÜÊNCIA>
		  <PROCESSAR ELEMENTO>
		  <ATUALIZAÇÃO DA VARIÁVEL SENTINELA>
		}

Repetição com indicador de passagem	Você quer repetir um conjunto de ações enquanto uma condição for verdadeira. Em geral, o conjunto de ações está relacionado ao processamento de uma seqüência de elementos. Porém, durante esse processamento, você precisa verificar uma determinada propriedade da seqüência para: (1) interromper o processamento e/ou (2) para, após o término da repetição, realizar uma ação que dependa daquela propriedade. Para isso, uma variável, chamada de **indicador de passagem**, é usada para indicar se a propriedade foi ou não satisfeita.	**int** indica;
		<INICIALIZAÇÕES>
		indica=0;
		
		while (<CONDIÇÃO DA REPETIÇÃO>&&INDICA ==0){
		  <LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQÜÊNCIA>
		  <PROCESSAR ELEMENTO>
		  if (<CONDIÇÃO DA VARIÁVEL INDICADOR DE PASSAGEM>
		  indica=1
		}
		if(indica == 1){
		  <CONJUNTO-DE-AÇÕES>
		else{
		  <CONJUNTO-DE-AÇÕES>
		}

Tabela 4.3: Exemplos de Padrões Elementares

Motivação	Processar uma seqüência de elementos que são números. Os elementos podem ser lidos (entradas do programa) ou gerados. Processar uma seqüência pode ser: a) executar ações na seqüência, por exemplo, contar seus elementos, somar ou modicá-los ou b) verificar algumas propriedades sobre a própria seqüência ou subseqüências; para verificar propriedades sobre elementos simples ou entre elementos.

Tabela 4.4: Motivação para os Padrões Elementares da Tabela 4.3

4.3 ProPAT: um plug-in Eclipse

PROPAT é parte do projeto Eclipse [Ecl] do Instituto de Matemática e Estatística da Universidade de São Paulo, financiado pela *IBM*© para a construção de um *Ambiente Integrado de Desenvolvimento* (IDE) para cursos de introdução à programação [2]. Neste ambiente, o estudante é capaz de escolher exercícios e construir uma solução selecionando e adicionando padrões de programação no editor. PROPAT também permite ao professor inserir novos padrões e exercícios.

O Projeto foi desenvolvido primeiramente para a linguagem C. Assim, algumas das características do plug-in PROPAT foram herdadas do plug-in Eclipse original CDT [Pi00], uma IDE para programação C, enquanto outras foram especialmente desenvolvidas para o projeto. Em termos gerais, um plug-in Eclipse é composto de perspectivas e visões (*views*). Sendo uma perspectiva um repositório para um conjunto de visões. Uma visão é usada, por exemplo, para abrir o editor, mostrar suas características e navegar em hierarquias de projetos, classes e conceitos.

O plug-in PROPAT herda algumas visões do plug-in Eclipse CDT original para compor duas novas perspectivas: a **Perspectiva do Aluno**: em que os alunos podem escolher exercícios e programar por meio da seleção e inserção de padrões, ou ainda podem escrever livremente seu próprio código; e a **Perspectiva do Professor**: usada pelo professor para especificar novos exercícios e padrões que estarão disponíveis posteriormente para o estudante na Perspectiva do Aluno. Essas duas perspectivas são descritas em mais detalhes a seguir.

Uma característica importante do plug-in PROPAT é a base de dados de padrões e exercícios. PROPAT usa uma base de dados XML para os padrões e exercícios que pode ser acessada pela comunidade de padrões pedagógicos..

[2] A primeira fase do projeto foi concluída em março de 2004 [Pro04]

nome do padrão *Repetição com sentinela*

intenção do padrão

Procesar uma seqüência com um número desconhecido de elementos e terminada com um valor dado.

intenção pedagógica

Fazer com que o aluno aprenda: (i) uma forma de processar uma seqüência sem o uso de um contador de elementos; (ii) usar um valor constante como indicador de fim de seqüência.

motivação

Processar uma seqüência de elementos que são números. Os elementos podem ser lidos (entradas do programa) ou gerados. Processar uma seqüência pode ser: (i) executar ações na seqüência, por exemplo, contar seus elementos, somar ou modificá-los; (ii) verificar algumas propriedades sobre a própria seqüência ou subseqüências; (iii) verificar propriedades sobre cada elemento ou entre elementos.

estrutura da solução 1: sintaxe

```
<INICIALIZAÇÕES>
<INICIALIZAÇÃO DA VARIÁVEL SENTINELA>
while        (<CONDIÇÃO DA VARIÁVEL SENTINELA>){
        <LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQUÊNCIA >
        <PROCESSAR ELEMENTO>
        <ATUALIZAÇÃO DA VARIÁVEL SENTINELA>
 }
```

O conjunto de ações é composto por:

<LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQUÊNCIA >,<PROCESSAR ELEMENTO> e <ATUALIZAÇÃO DA VARIÁVEL SENTINELA>

estrutura da solução 2: semântica

A inicialiação de variáveis e inicialização da sentinela para condição verdadeira são executadas uma única vez. Em seguida, a condição da sentinela é verificada;(1) um elemento da sequência é lido/gerado, (2) processado e, eventualmente, (3) a variável sentinela é atualizada.
Em seguida, a condição da variável sentinela é novamente verificada. O processo pára quando a condição da variável sentinela se tornar falsa.

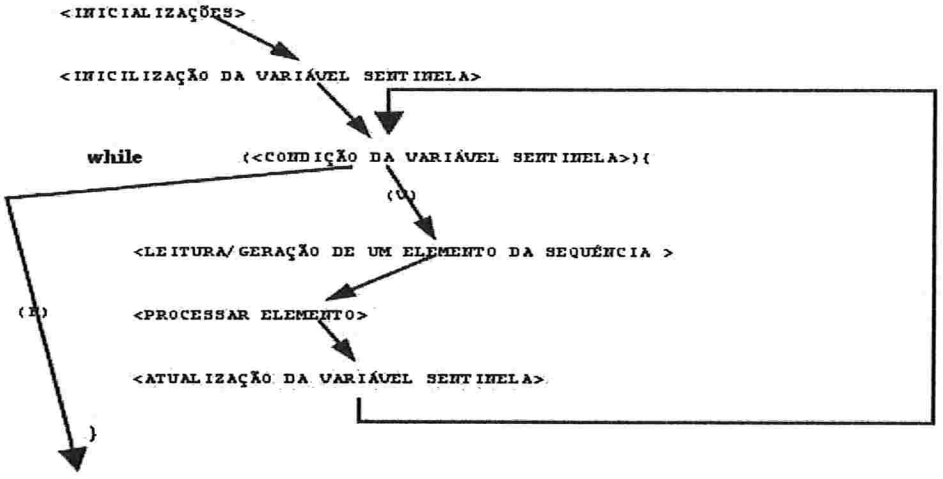


Figura 4.1: Padrão Repetição com sentinela

nome do padrão **Repetição com sentinela (continuação)**

aplicabilidade / objetivo

Você quer repetir um conjunto de ações, que em geral, está relacionado ao processamento de uma seqüência de elementos. Os elementos da seqüência podem ser lidos ou gerados. A quantidade de elementos é desconhecida mas o fim da seqüência é indicado por um valor sentinela. Isso quer dizer que, o número de repetições depende da variável sentinela e portanto, essa variável deve estar na condição da repetição.

pré-requisitos **Declaração de Variável Inteira, Conjunto de Ações, inicialização, leitura**

implementação

A atualização da variável sentinela e a condição da variável sentinela devem ser feitas corretamente, caso contrário, o programa pode entrar em um "laço infinito".

código exemplo Faça um programa que leia uma sequência de números inteiros terminada por zero e calcule a sua soma.

```
soma=0;
printf("Digite o numero inteiro: ");
scanf ("%d", &numero); /*leitura do primero numero */
while (numero != 0) {
        soma = soma + numero;
        printf ("Digite o numero inteiro: ");
        scanf("%d", &numero);  /*leitura do numero seguinte*/
}
```

usos conhecidos

---

padrões relacionados **Repetição contada e Repetição com indicador de passagem**

Figura 4.2: Padrão Repetição com sentinela continuação

4.3.1 Perspectiva do Aluno

Existem oito visões disponíveis nesta perspectiva (Figura 4.3): **Program Editor *View***, **Navigator *View***, **Pattern *View***, **Pattern Info *View***, **Metadata *View***, **Exercise *View***, **Exercise Description *View*** e **Message Console**.

- **Pattern View** (parte direita da Figura 4.3) contém uma lista de todos os padrões pedagógicos armazenados na base de dados XML. Essa visão está sincronizada com o **Pattern Info *View*** (parte inferior da Figura 4.3) de modo que o aluno pode selecionar o padrão no **Pattern View** e observar a sua documentação (conforme a Tabela 4.2 no **Pattern Info View**).
- Usando o Pattern View, o estudante pode selecionar e inserir o código da Sintaxe de um padrão no **Program Editor *View*** (parte central da Figura 4.3) diretamente em seu programa. Esse código pode conter expressões entre apóstrofes (metadados) que o estudante pode substituir por outros padrões ou seu próprio pedaço de código. É interessante notar que o plug-in não deixa o aluno inserir padrões em posições não permitidas pelas regras de sintaxe da linguagem (por exemplo, da linguagem C). Se o estudante tenta fazer isso, uma mensagem de erro é gerada. Com isso, o estudante poderá aprender algumas regras simples de sintaxe da linguagem de programação.
- **Metadata *View*** permite ao estudante navegar na lista de metadados que tem sido usados pelo conjunto de padrões. Quando selecionamos os nomes da lista, uma caixa de texto mostra a definição do metadado. Esperamos que os metadados ajudem o aluno a aprender novos conceitos de programação como: *inicialização* ou *atualização do contador*.
- **Exercise *View*** é similar ao Pattern View. Essa visão contém uma lista com os nomes de todos os exercícios disponíveis na base de dados XML, organizados por tópicos de programação. Cada exercício está relacionado com um ou mais padrões. O desafio do estudante é encontrar os padrões mais adequados para resolver um problema.
- **Exercise Description *View***, mostra ao estudante a descrição do exercício selecionado e contém o nome do exercício, a especificação do problema e o conjunto de casos de teste.

A perspectiva do estudante é organizada de modo que o editor do programa e todas suas visões sejam visualizadas em uma única janela. Além disso, na perspectiva do aluno existe um botão para compilar e executar o programa, sendo fácil que o estudante teste suas soluções na visão **Message Console**, usando os casos de teste fornecidos na descrição dos exercícios.

4.3.2 Perspectiva do Professor

A **Perspectiva do Professor** (Figura 4.4) torna o PROPAT uma ferramenta de autoria, isto é, uma ferramenta de LMS cujo conteúdo pode ser preenchido pelo professor, provendo ajuda ao professor que usa

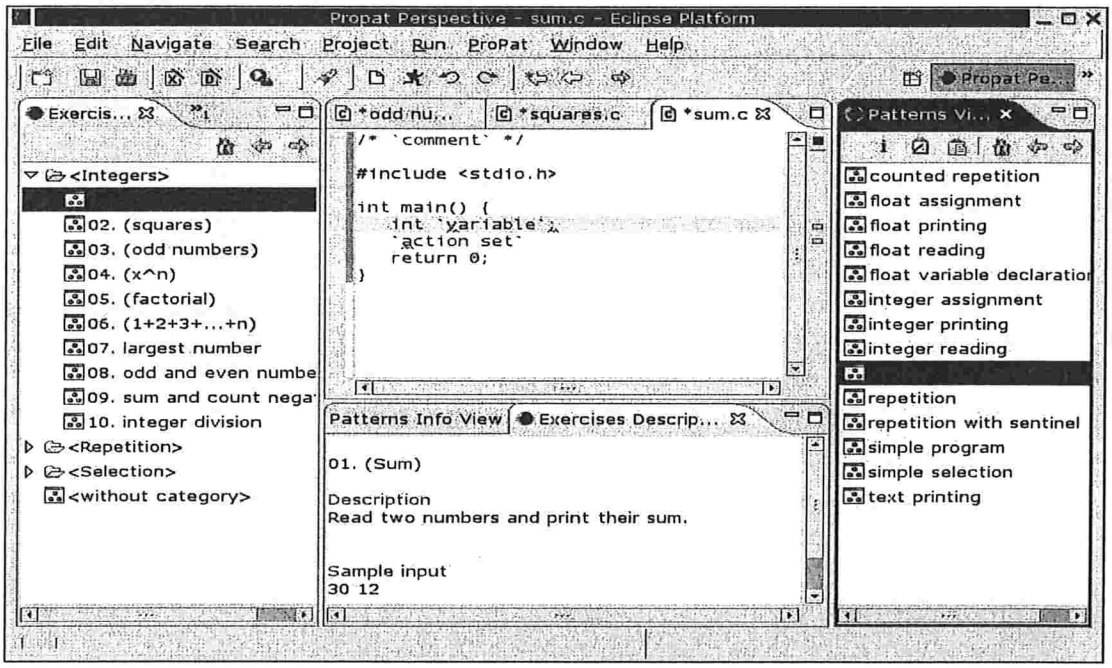


Figura 4.3: Perspectiva do aluno de PROPAT-Pattern View, Pattern Info View e Exercise View

padrões pedagógicos de programação em suas disciplinas.

Esta perspectiva herda algumas características disponíveis na Perspectiva do Aluno e adiciona algumas visões necessárias para editar, remover e inserir novos padrões e exercícios. Para esse propósito foram criadas três *views* adicionais: **Pattern Editor View**, **Exercise Editor View** e **Metadata Editor View**.

- Na **Pattern Editor *View***, para editar e criar um padrão, o professor deve preencher um formulário, que corresponde à descrição do padrão. Além disso, existe uma informação extra que o professor deve prover: as restrições de inserções no padrão, chamadas também de regras de inserção de padrões. Essa informação é usada pela **Program Editor *View*** para checar alguns erros do estudante e dar alguns avisos.

- Os exercícios têm também dois novos campos: o exemplo de solução do programa (não acessível pelo estudante) e os padrões sugeridos pelo professor, que ele considera a melhor escolha para resolver o problema. Os campos podem ser editados pelo professor por meio do **Exercise Editor *View***. Um dos usos da solução para o exercício dada pelo professor é executá-lo com os casos de teste para calcular os valores esperados das saídas.

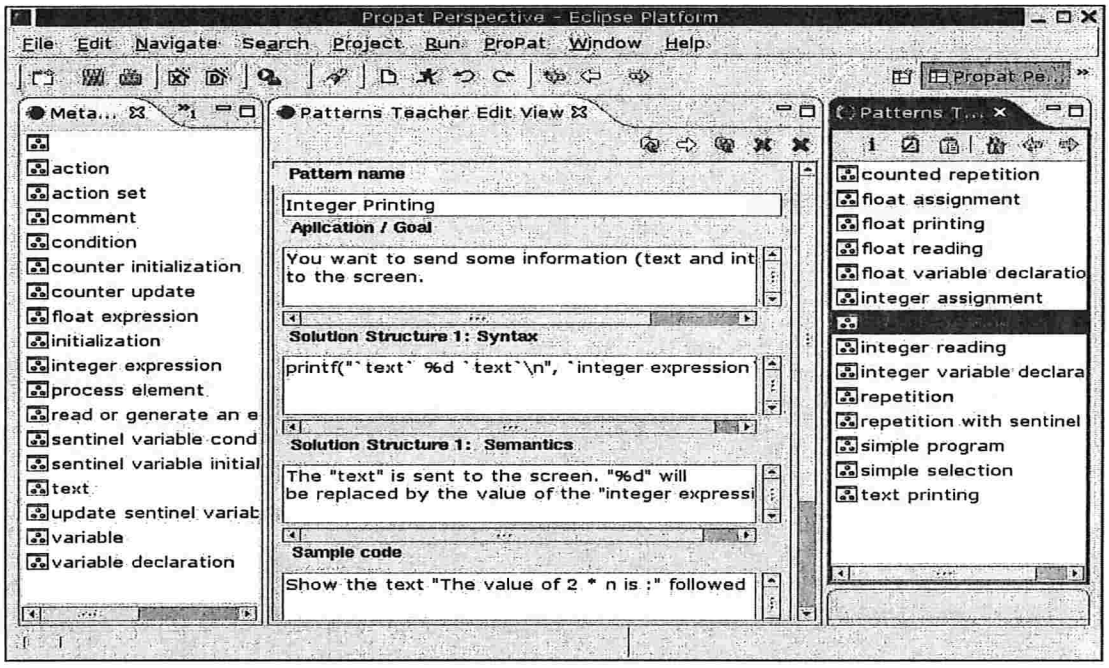


Figura 4.4: Perspectiva do professor de PROPAT-Pattern View e Pattern Editor View

- O **Metadata Editor** ***View*** é um editor de texto que permite ao professor definir metadados, através de um texto mostrado ao estudante e aos professores. Esta visão incentiva ao professor usar os mesmos metadados, sempre que possível, enquanto cria novos padrões.

4.3.3 O sistema ProPAT e um paralelo com os sistemas tutores de programação

O sistema PROPAT, proposto neste trabalho, é baseado no PROUST. Apesar do sistema PROUST ter sido usado como base para muitos trabalhos que o sucederam, incluindo o sistema PROPL, ele possui algumas limitações que o impediram de ser usado, efetivamente, no ensino de programação, entre elas:

1. para que o diagnóstico seja correto, é necessário que exista uma biblioteca completa de planos especificados por educadores experientes de programação, uma vez que aqueles planos devem corresponder a todas as maneiras possíveis que estudantes aprendizes resolvem problemas;
2. professores nem sempre concordam sobre quais planos devem ser incluídos na biblioteca porém, na proposta original do sistema, não é possível adicionar novos elementos na biblioteca de planos;

3. os planos são representados internamente, sem que os alunos possam ter acesso, deixando para a interface de comunicação do sistema totalmente responsável por promover o aprendizado do aluno.

No sistema PROPAT, a biblioteca de planos é substituída por uma biblioteca de Padrões Elementares (Seção 4.1.1) que, ao contrário do que foi proposto em PROUST, pode ser acessada diretamente tanto pelo aluno como pelo professor, funcionando como um material de estudos e aumentando a probabilidade dos alunos empregarem os padrões em seus programas. Os padrões poderão ser definidos pelo próprio professor e, ao contrário de PROUST, eles não precisam corresponder a todas as formas corretas de resolver problemas mas sim, às formas didáticas recomendadas por educadores de programação.

O programa final, construído pelo aluno, será diagnosticado por um sistema do tipo Diagnóstico Baseado em Modelos (Seção 5) que, baseado nas inserções de padrões feitas pelo aluno (monitoradas por uma interface ou identificadas por meio de perguntas feitas ao aluno), será capaz de estabelecer um diálogo de mais alto nível durante o diagnóstico.

O diagnóstico de programas proposto para o sistema PROPAT, implementa uma nova combinação das abordagens descritas na Seção 3.1:

- **Verificação com respeito ao conhecimento da linguagem**: verificação do emprego dos padrões de programação disponibilizados pelo professor para resolver o problema dado.
- **Filtragem com respeito aos sintomas** usando MBD: analisa um modelo do programa do aluno representado na forma de componentes e conexões, onde os componentes correspondem às estruturas lógicas da linguagem e também aos padrões de programação.

A Tabela 4.5 contém os exemplos mais citados de sistemas tutores (alguns deles não foram descritos em detalhes por apresentarem características muito semelhantes aos já descritos na Seção 3.3.1) em ordem cronológica de suas publicações juntamente com as técnicas que eles empregam para correção automática. Note que o módulo de diagnóstico do PROPAT, que chamaremos de PROPAT_DEBUG, implementa uma nova combinação de abordagens: verificação com respeito ao conhecimento da linguagem e MBD.

4.4 Considerações finais

Nesse capítulo foi introduzido o conceito de *Padrões Elementares* de programação que podem ser vistos como estratégias gerais de solução para problemas recorrentes. Padrões Elementares podem ser usados para identificar as intenções do aluno nos programas e para estabelecer uma melhor comunicação entre professor/-ferramenta e aluno. Foi descrita também a ferramenta PROPAT, um ambiente integrado de desenvolvimento em que o estudante pode aprender a programar usando *Padrões Elementares*. Neste ambiente após compilar o programa, este pode ser testado e em caso de falha o sistema de diagnóstico PROPAT_DEBUG pode ser chamado.

Nome do Tutor Inteligente de Programação	**Verificação com respeito à especificação**	**Verificação com respeito ao conhecimento da linguagem**	**Outras técnicas de Filtragem**	**MBD**
INTELLIGENT PROGRAM ANALYSIS [1976]	X			
PUDSY [1980]	X	X		
LAURA [1980]	X			
PROUST [JS84]	X	X	X	
LISP TUTOR [AS86]	X			
PHENARETE [1987]		X		
TALUS [1988]	X			
APROPOS [1988]	X	X		
KUMAR'S TUTOR [Kum02]				X
PROPAT [2004]		X		X

Tabela 4.5: Alguns dos Tutores Inteligentes de Programação mais citados e técnicas que eles implementam para correção automática.

No próximo capítulo será detalhada a técnica geral de Diagnóstico Baseado em Modelos (MBD). No Capítulo 6, será descrita a técnica de DBM para diagnóstico de programas e a especificação do sistema PROPAT_DEBUG.

CAPÍTULO 5

Diagnóstico Baseado em Modelos de Dispositivos Físicos

Em termos gerais, o diagnóstico pode ser descrito como o processo de inferir explicações para um conjunto de observações. Existem duas abordagens para a tarefa de diagnóstico em Inteligência Artificial [Moz92]:

- *Diagnóstico Heurístico*, baseado na experiência, abordagem em que o projetista do sistema deve interagir com o especialista para obter a informação de como fazer o diagnóstico. A informação normalmente está na forma de regras do tipo "se sintoma então falha". Dizemos que esse é o conhecimento do especialista em diagnóstico "compilado" na forma de heurísticas. Em consequência, estes sistemas de diagnóstico são altamente especializados e restritos a aplicações que requerem experiência no domínio, acarretando custo elevado para desenvolvimento e manutenção.

- *Diagnóstico Baseado em Modelos*, também chamado de *diagnóstico baseado em princípios básicos*, não requer conhecimento do especialista em diagnóstico, mas sim a descrição do projeto do sistema. O primeiro sistema MBD (Model Based Diagnosis) foi desenvolvido por de Kleer em 1976 [dW76]. O paradigma básico de MBD pode ser entendido como a interação de observações e predições (Figura 5.1) [Ben93]. Por um lado, temos um sistema real cujo comportamento pode ser observado. Por outro lado, temos o modelo do sistema que pode ser usado para fazer predições sobre seu comportamento. A diferença entre a observação e a predição é chamada *discrepância*. A equivalência entre a observação e a predição é chamada *concordância*. Discrepâncias e concordâncias são usadas para identificar as partes falhas do sistema [Ben93].

As categorias de modelos que podem ser usados em MBD são:

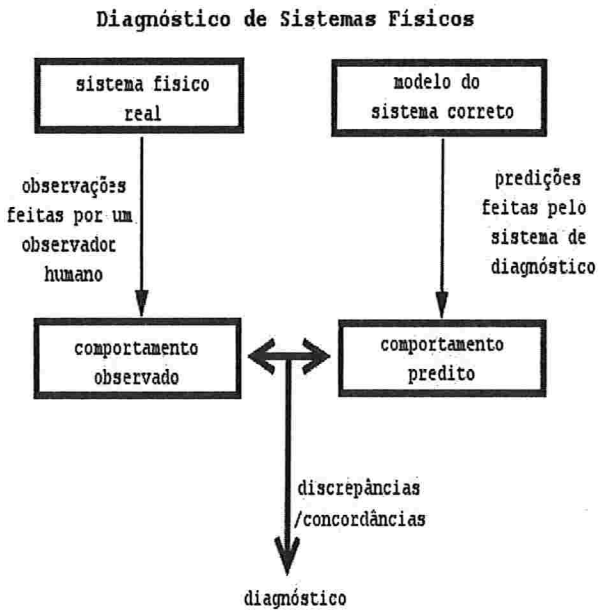


Figura 5.1: Diagnóstico Baseado em Modelos de sistemas físicos, visto como a interação de observações e predições.

1. **Modelos Centrados em Componentes:** descrevem o sistema pelos seus componentes. Exemplos de modelos centrados em componentes são o modelo estrutural e comportamental dos componentes;

2. **Modelos Causais:** descrevem o sistema em termos de relação entre estados do tipo causa e efeito, isto é, estados causam outros estados como efeito.

3. **Modelos Funcionais:** descrevem o sistema em termos de funções e sub-funções.

O Modelo Centrado em Componentes representa explicitamente a estrutura do sistema em termos de componentes e conexões (*modelo estrutural*). Além disso, o modelo baseado em componentes descreve o comportamento do sistema com relação às suas entradas e saídas (*modelo comportamental*).

Um sistema MBD que usa o Modelo Centrado em Componentes pode usar o modelo do comportamento correto ou falho do sistema. Se o comportamento correto é usado, uma falha é caracterizada pela ausência de comportamento correto. De acordo com o tipo de inferência que o sistema de diagnóstico faz sobre o modelo, este pode ser classificado em: *baseado em consistência* ou *abdutivo* [Ben93].

Na Tabela 5.1 definimos alguns termos usados nessa seção e que serão definidos formalmente na Seção 5.2.

SD	Descrição do sistema (*Sistem Description - SD*).
COMP	Conjunto de componentes do sistema.
OBS	Observação de um sistema.
Δ	Diagnóstico minimal para (SD,COM,OBS).
CO	Conjunto dos contribuintes de uma falha.
AB(C)	Especifica o comportamento anormal do componente $C \in COMP$.
$\neg AB(C)$	Especifica o comportamento normal do componente $C \in COMP$.
CONNECT	Conjunto de conexões do sistema.

Tabela 5.1: Alguns termos usados em Diagnóstico Baseado em Modelos.

Em sistemas que aplicam o método abdutivo, a solução do diagnóstico tem a característica que todas as observações são conseqüência lógica da solução do diagnóstico, isto é, a descrição do sistema junto com os componentes que tem comportamento anormal (que pertencem ao diagnóstico) e os componentes que tem comportamento normal (que não pertencem ao diagnóstico), têm como conseqüência lógica as observações:

$$SD \cup \{AB(C)|C \in \Delta\} \cup \{\neg AB(C)|C \in COMP \setminus \Delta\} \vdash OBS$$

Por outro lado, em sistemas baseados em consistência, a solução do diagnóstico deve ser consistente com as observações, isto é, a descrição do sistema junto com as observações e os componentes que tem comportamento anormal (que pertencem ao diagnóstico) e os componentes que tem comportamento normal (que não pertencem ao diagnóstico) são logicamente consistentes:

$$SD \cup OBS \cup \{AB(C)|C \in \Delta\} \cup \{\neg AB(C)|C \in COMP \setminus \Delta\} \nvdash \bot$$

O tipo de diagnóstico baseado em modelos implementado nesse trabalho é baseado em consistência e usa modelos corretos, comportamental e estrutural (abordagem centrada em componentes).

5.1 Modelo Centrado em Componentes

O Diagnóstico Baseado em Modelos Centrados em Componentes, descreve o sistema por meio de um modelo estrutural e comportamental. No modelo comportamental existem dois tipos de regras [Ben93]:

- Regras de simulação (para frente ou progressivas): predizem os valores para as saídas de um componente, dadas as entradas. Podem descrever o comportamento correto/normal dos componentes.
- Regras de inferência (para trás ou regressivas): regras que podem não descrever o comportamento correto

real do sistema (valores que se propagam da saída para a entrada), mas descrevem conclusões válidas sobre o comportamento do sistema pois calculam as possíveis entradas do componente, dadas as saídas.

Existem duas propostas de sistemas que usam o Modelo Centrado em Componentes, que são descritas na Seção 5.5: *HT* [Dav84] e *General Diagnosis Engine - GDE* [dW87].

5.2 Diagnóstico Baseado em Consistência

Raymond Reiter [Rei87] propõe a abordagem do diagnóstico baseado em consistência cuja formulação do problema é apresentada a seguir.

Definição 5.1: [Rei87]: O *sistema de diagnóstico* é um par (SD, COMP) em que:

- COMP é o conjunto de *componentes do sistema*, representado por um conjunto finito de constantes.
- SD é a *descrição do sistema (o modelo)*, dada por um conjunto finito de sentenças de primeira ordem (para modelos centrados em componentes, essas sentenças descrevem as regras de simulação e de inferência dos componentes do sistema). A descrição do sistema é dada em termos do conjunto COMP e as conexões entre os componentes $C \in COMP$. Uma vez que usaremos o modelo correto do sistema, as sentenças em SD devem descrever o comportamento correto dos componentes. Assim, $\neg AB(C)$ indica que o componente C está correto (*NOT_ABNORMAL(C)*).

  CONNECT é o conjunto formado por triplas $< C_i, C_j, V >$, em que $C_i \in COMP$, $C_j \in COMP$ e V é o valor dado pela relação $V : COMPxCOMP \rightarrow R$. [1]

  Por exemplo, na Figura 5.4 (descrita em mais detalhes na Seção 5.4), o componente $COMP_X \in COMP$ pode ser descrito com base no comportamento correto de uma porta lógica AND ($\neg AB(COMP_X)$). As regras de inferência que relacionam as conexões de entrada P e C, com a conexão de saída I são:

  $P \wedge C \wedge \neg AB(COMP_X) \rightarrow I$ $\qquad$ $\neg(P \wedge C) \wedge \neg AB(COMP_X) \rightarrow \neg I$

  Para o mesmo componente, mostramos a seguir algumas regras de simulação:

  $C \wedge \neg I \wedge \neg AB(COMP_X) \rightarrow \neg P$

  $C \wedge I \wedge \neg AB(COMP_X) \rightarrow P$

  $P \wedge \neg I \wedge \neg AB(COMP_X) \rightarrow \neg C$

  $P \wedge I \wedge \neg AB(COMP_X) \rightarrow C$

Definição 5.2 [Rei87]: A *observação* de um sistema, OBS, é um conjunto finito de sentenças de primeira ordem. Por exemplo, na Figura 5.4, uma observação poderia incluir valores para as conexões numa situação

[1] Os tipos dos valores das conexões podem ser: booleanos, reais, inteiros ou caracteres, isso depende do sistema. Por simplificação nós assumimos valores reais.

em particular. A tripla (SD, COMP, OBS) é chamada de um *problema de diagnóstico* para o sistema (SD, COMP) com observações OBS.

Definição 5.3 [Rei87]: Dado $C \in COMP$, $AB(C)$ significa que o componente C possui um comportamento anormal, isto é, que não está de acordo com a descrição do sistema (SD). Um *diagnóstico* para (SD, COMP, OBS) é um conjunto $\Delta \subseteq COMP$ tal que $SD \cup OBS \cup \{AB(C)|C \in \Delta\} \cup \{\neg AB(C)|C \in COMP \setminus \Delta\}$ é consistente. Um diagnóstico é chamado de minimal se nenhum subconjunto dele é também um diagnóstico.

Definição 5.4 [Rei87]: Um *conjunto de contribuintes* para (SD, COMP, OBS) é um conjunto $CO \subseteq COMP$ tal que $SD \cup OBS \cup \neg AB(C)|C \in CO$ é inconsistente.

Definição 5.5 [Rei87]: Seja C uma coleção de conjuntos. Um *conjunto de corte (hitting set)* para C é um conjunto

$$H \subseteq \bigcup_{S \in C} S$$

tal que $\forall S \in C, H \cap S \neq \varnothing$ (o conjunto de corte é um conjunto que intercepta todos os conjuntos da coleção). Um conjunto de corte é minimal se nenhum subconjunto dele é também um conjunto de corte.

O seguinte teorema mostra uma abordagem construtiva para encontrar o diagnóstico a partir de uma coleção de conjuntos de contribuintes.

Teorema 5.1 [Rei87]: *O conjunto $\Delta \subseteq COMP$ é um diagnóstico minimal para (SD, COMP, OBS) se Δ é um conjunto de corte minimal para a coleção de conjuntos de contribuintes.*

5.3 Tarefas do processo de diagnóstico

O raciocínio de diagnóstico pode ser concebido como a execução de três tarefas [Ben93] (Figura 5.2): *detecção de sintomas, geração de hipóteses* e *discriminação de hipóteses.*

5.3.1 Detecção de Sintomas

O diagnóstico tem como entrada as observações iniciais, as quais podem ser chamadas de *sintomas.* Sintomas são *observações anormais*, isto é, uma observação que difere do esperado. As observações corretas são definidas como *observações normais.*

5.3.2 Geração de Hipóteses

Dado um conjunto de sintomas, a geração de hipóteses sugere possíveis hipóteses (causas) para aqueles sintomas, com base nas observações iniciais. Uma hipótese corresponde a um ou mais componentes que podem estar falhos e que expliquem os sintomas.

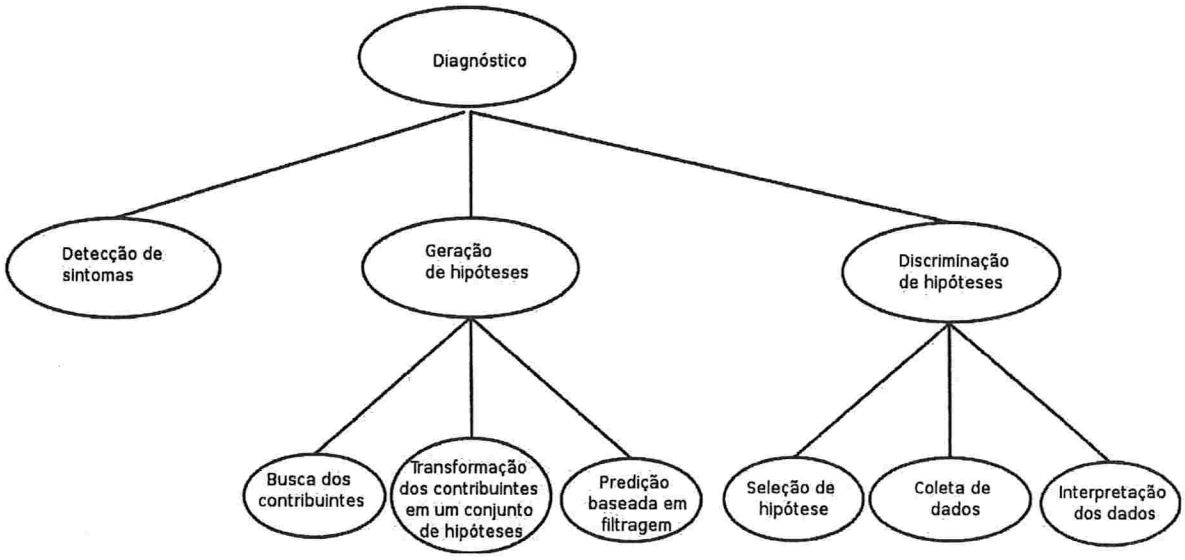


Figura 5.2: Decomposição do processo de diagnóstico em 3 tarefas

A tarefa de geração de hipóteses pode ser decomposta em três sub-tarefas:

- **Busca dos contribuintes:** o conjunto de contribuintes é um conjunto de componentes que contém pelo menos um componente incorreto. O conjunto de contribuintes é também chamado de *conjunto conflito* [dW87].

- **Transformação dos contribuintes em um conjunto de hipóteses:** os conjuntos de contribuintes são transformados em um conjunto de hipóteses. Todas as hipóteses no conjunto de hipóteses explicam todas as observações iniciais, isto é, se todos os componentes de uma hipótese são assumidos incorretos, a descrição do sistema e as observações são consistentes (Ver Definição 5.3 da Seção 5.2). Uma das formas de transformar os conjuntos de contribuintes em um conjunto de hipóteses é aplicar o método de cobertura de conjunto (*set covering*), que executa uma operação sobre os conjuntos. A idéia é que uma hipótese não deve ter intersecção vazia com nenhum dos conjuntos contribuintes, ou seja, uma hipótese cobre todos os conjuntos de contribuintes. Os conjuntos de corte minimais para a família de contribuintes calculados pelo algoritmo de Reiter [Rei87] correspondem ao resultado de cobrir todos os conjuntos de contribuintes. Assim, o algoritmo de Reiter (Figura 5.3) pode ser usado para esta tarefa.

- **Predição baseada em filtragem:** o processo de filtragem é usado para excluir hipóteses usando somente o conhecimento já existente sobre o sistema (sem necessidade de observações adicionais). As hipóteses restantes após a filtragem deverão ainda ser discriminadas pela tarefa de *discriminação de*

hipóteses (Seção 5.3.3).

O Algoritmo de Reiter

O Algoritmo de Reiter é usado para transformar os contribuintes em um conjunto de hipóteses (Figura 5.3). Reiter [Rei87][2], descreve o algoritmo que calcula os conjuntos de corte minimais para uma família de conjuntos F.

O algoritmo gera um grafo acíclico com nós rotulados por conjuntos de contribuintes e arcos rotulados por elementos do conjunto. O grafo é expandido em largura. A idéia é que para cada nó rotulado por um conjunto de contribuintes S, os arcos saindo dele são rotulados pelos elementos de S (Figura 5.5).

O rótulo do primeiro nó do grafo é gerado chamando-se um provador de teoremas para provar que a entrada: $SD \cup COMP \cup OBS$, é satisfazível. Dado que existe uma inconsistência entre o modelo e as observações, o provador de teoremas devolverá o conjunto de sentenças envolvidas na inconsistência encontrada (I). Como essas sentenças incluem componentes e conexões, é preciso selecionar apenas os componentes candidatos à falha [3] fazendo-se $CO = I \cap COMP$, o que no início da busca resulta no rótulo do nó raiz da árvore de busca. Caso o provedor de teorema consiga satisfazer a entrada, isto é, nenhum conjunto de contribuintes foi encontrado, o nó é rotulado como @. Por exemplo, na Figura 5.5, o primeiro nó da busca é rotulado por $S = \{COMP_X, COMP_Y, COMP_W, COMP_Z, COMP_V\}$, sendo esse o conjunto de componentes candidatos à falha (conjunto de contribuintes).

A árvore é então expandida em largura onde cada arco é rotulado com um elemento $s \in S$. Para rotular um novo nó a partir desse arco, o provador de teoremas é chamado novamente considerando o componente s como falho ($AB(s)$). Como a descrição do sistema (SD) possui apenas sentenças para $\neg AB(s)$, os valores das conexões de entrada e saída de s não serão propagados através do componente s. Isso define umas das técnicas de geração de hipóteses chamada de *suspensão*. Por exemplo, na Figura 5.5 (parte a), o nó rotulado por $S = \{COMP_X, COMP_W, COMP_Z, COMP_V\}$ foi gerado pela chamada ao provador de teoremas com a entrada $SD \cup (COMP \setminus \{COMP_Y\}) \cup OBS$, isto é, fazendo-se a suspensão do componente $COMP_Y$. Já a suspensão do componente $COMP_X$ (Figura 5.5, parte a) resultou num conjunto de contribuintes vazio (nó rotulado por @) e portanto $COMP_X$ é uma hipótese de falha única que explica as observações.

Um aspecto interessante do algoritmo de Reiter é que ele permite encontrar falhas múltiplas. Por exemplo, na Figura 5.5 c), o nó $\{COMP_X, COMP_W, COMP_Z\}$ foi gerado fazendo-se a suspensão de dois componentes, $COMP_Y$ e $COMP_V$.

Seja H(n) o conjunto formado pelas rótulos dos arcos no caminho da raiz (componentes suspendidos) até o nó n. O nó n é rotulado por um conjunto S tal que $S \cap H(n) = \varnothing$. Qualquer caminho terminado com um

[2] O algoritmo originalmente proposto por Reiter, sofreu algumas correções feitas por [GSW89]

[3] Uma conexão também pode estar falha mas nesse algoritmo considera-se apenas a possibilidade dos componentes estarem falhos.

nó rotulado por @ é um conjunto de corte, dado que ele intercepta todos os possíveis rótulos dos nós.

O algoritmo de Reiter é um algoritmo geral[4] para encontrar um conjunto de corte de uma família de conjuntos quaisquer F. Para isso, o algoritmo tenta gerar a menor quantidade possível de novos nós com rótulos que correspondem aos elementos de F (conjuntos), que não são conhecidos a priori. Ou seja, os elementos de F são conjuntos de candidatos gerados pelas sucessivas chamadas a um provador de teoremas, como foi ilustrado anteriormente para a tarefa de diagnóstico.

Uma vez que calcular um elemento de F é uma operação muito cara, o algoritmo minimiza o número de chamadas fazendo poda ao grafo em três passos do algoritmo (Figura 5.3):

1. **Passo 2.c:** poda de sub-árvores repetidas;

2. **Passo 2.d:** quando o conjunto de rótulos dos arcos de um caminho é um super-conjunto de outro caminho que termina em @, faz-se a poda do caminho maior para assim gerar hipóteses minimais;

3. **Passo 2.f.i:** Se um rótulo de nó S é um sub-conjunto de outro rótulo S' então todos os caminhos a partir do super-conjunto, rotulados com a diferença entre o super-conjunto e o sub-conjunto ($S^* \setminus S'$) devem ser removidos (Figura 5.5, parte c).

Teorema 5.2 [Rei87]: *Os cortes minimais de F são {H(n) tal que n tem rótulo @}*

A Figura 5.3 descreve o algoritmo de Reiter. De acordo com o Teorema 5.2, o algoritmo devolve um conjunto hipótese (corte minimal) sempre que encontra um nó rotulado por @, ou seja, quando faz uma chamada ao provador de teoremas sem encontrar uma inconsistência.

Note que o algoritmo de Reiter faz chamadas sucessivas a um provador de teoremas para recuperar o conjunto conflito envolvido numa inconsistência. No entanto, não detalhamos aqui como isso pode ser implementado. Uma das técnicas usadas para encontrar as sentenças envolvidas na inconsistência é o uso de ATMS (*Assumption-Based Truth Maintenance System*) [de 86]. Um ATMS é usado para determinar as suposições feitas por um provador de teoremas sobre um sistema ou uma teoria sobre um domínio. Além disso, um ATMS é usado para registrar as inferências feitas durante uma prova que servirá, eventualmente, como justificativas para um usuário ou, no caso do uso do algoritmo de Reiter, para recuperar o conjunto conflito.

Na Seção 6.4 do Capítulo 6 será mostrada uma implementação para a geração do conjunto de contribuintes baseada em sistemas de produção com registro de dependências.

[4]O algoritmo de Reiter é também usado na área de Revisão de Crenças para representação de conhecimento [Was99].

ReiterAlgorithm(F)

1. Escolha um elemento de F para rotular a raiz (nível 0)
2. **Para cada** nó n do nível i **faça:**
   (a) **Se** n tem rótulo S **então** para cada $s \in S$ crie um *s_arco* saindo de n com rótulo s.
   (b) H(n)= conjunto de rótulos do caminho da raiz até n.
   (c) **Se** para algum n' temos $H(n') = H(n) \cup \{s\}$ **então** o *s_arco* aponta para n'
   (d) **Se** $\exists n'$ com rótulo @ tal que $H(n') \subset (H(n) \cup \{s\}$ **então** fecha o s_arco.
   (e) **Senão se** $\exists n'$ com rótulo S' e $S' \cap (H(n) \cup \{s\}) = \varnothing$ **então** o s_arco aponta para um novo nó com rótulo S'.
   (f) **Senão** faça o s_arco apontar para um novo nó m e rotule m com o primeiro elemento S' de F tal que $S' \cap H(m) = \varnothing$ (para a tarefa de diagnóstico isto poderia ser feito por exemplo, com ajuda do provador de teoremas). **Senão** existir, use um rótulo especial @
      i. **Se** existe n' com rotulo S* tal que $S' \subset$ S* **então** mude o rótulo de n' para S' e remova todos os arcos partindo de n' com rótulo em S*$\setminus S'$
3. **Repete** o passo 2 para o nível i+1

Figura 5.3: Algoritmo de Reiter [Rei87] [GSW89].

5.3.3 Discriminação de hipóteses

O objetivo desta tarefa é podar o conjunto de hipóteses com base em observações adicionais (testes), de modo que reste apenas uma hipótese que é a solução para o problema, se possível. A tarefa de discriminação inclui:

- **Seleção de hipótese**: pode ser executada de uma maneira simples, por exemplo, selecionar uma hipótese de maneira aleatória ou selecionar a primeira hipótese encontrada. Pode se aplicar também uma estratégia mais refinada para ordená-las usando conhecimento adicional, por exemplo, custos de teste. A escolha da ordem de seleção de hipótese deve permitir encontrar a resposta tentando minimizar o número de testes subseqüentes.
- **Coleta de dados**: uma vez selecionada a hipótese, é feita a coleta dos dados, isto é, são feitas novas

observações, substituindo-se o(s) componente(s) ou executando-se testes diretos na saída da hipótese selecionada.

- **Interpretação dos dados**: o passo final é interpretar o dado obtido com respeito ao conjunto de hipóteses. Para isso, os métodos *isolamento*, *particionamento* ou *geração de hipóteses baseado em modelos* podem ser aplicados:
  - **Isolamento** é um método simples de interpretar e consiste em determinar o impacto local de uma observação, isto é, se a observação casar com o valor esperado ela pode ser descartada do conjunto de hipóteses, caso contrário ela continua.
  - Outro método usado para interpretar o dado obtido, chamado de **particionamento**, consiste em explorar as dependências entre as hipóteses. Assim, se a observação casar com o valor esperado, a hipótese propriamente dita junto com todas as hipóteses dependentes dela são descartadas.
  - O método de **geração de hipóteses baseado em modelos**, consiste em executar o mesmo processo de geração de hipóteses usado anteriormente para explicar o conjunto inicial de observações junto com as observações adicionais.

5.4 Exemplo de aplicação do processo de diagnóstico

Considere um sistema de alarme de carro que avise, tocando uma buzina, quando o carro está sendo dirigido com uma porta aberta ou quando está estacionado com as luzes acesas. O circuito de tal sistema é mostrado na Figura 5.4.

Seja:

- L, uma variável com valor 1 quando as luzes estão acesas;
- C, uma variável com valor 1 quando a chave está ligada;
- P, uma variável com valor 1 quando qualquer porta do carro estiver aberta;
- P0, P1, P2 e P3, variáveis que indicam se cada uma das 4 portas do carro estão abertas;
- A, uma variável com valor 1 se o alarme estiver acionado;

- SD, a descrição do sistema dada pelas seguintes sentenças da lógica de primeira ordem:

$$P \wedge C \wedge \neg AB(COMP_X) \rightarrow I \qquad \neg(P \wedge C) \wedge \neg AB(COMP_X) \rightarrow \neg I$$
$$C \wedge \neg AB(COMP_Y) \rightarrow \neg K \qquad \neg C \wedge \neg AB(COMP_Y) \rightarrow K$$
$$K \wedge L \wedge \neg AB(COMP_W) \rightarrow J \qquad \neg(K \wedge L) \wedge \neg AB(COMP_W) \rightarrow \neg J$$
$$(I \vee J) \wedge \neg AB(COMP_Z) \rightarrow A \qquad \neg(I \vee J) \wedge \neg AB(COMP_Z) \rightarrow \neg A$$

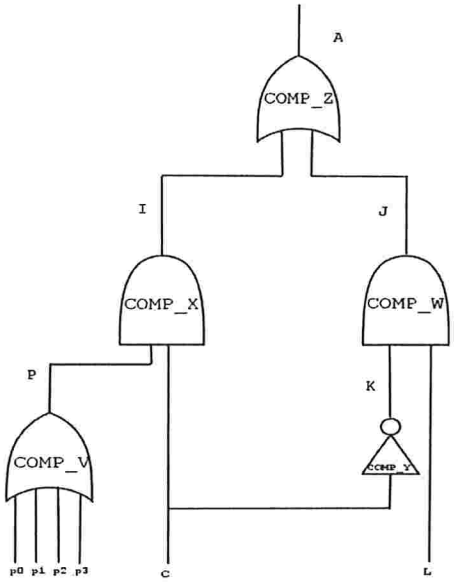


Figura 5.4: Circuito de um alarme de carro com o componente COMP_W falho.

$$(P0 \vee P1 \vee P2 \vee P3) \wedge \neg AB(COMP_V) \rightarrow P \qquad \neg(P0 \vee P1 \vee P2 \vee P3) \wedge \neg AB(COMP_V) \rightarrow \neg P;$$

- COMP, o conjunto de componentes do sistema composto pelas portas lógica OR, AND e NOT do circuito rotuladas por: $COMP_X$, $COMP_Y$, $COMP_W$, $COMP_Z$, $COMP_V$, ou seja, COMP={ $COMP_X$, $COMP_Y$, $COMP_W$, $COMP_Z$, $COMP_V$};

- OBS, o conjunto de observações iniciais. Na situação inicial, o alarme está acionado, com o carro estacionado, as luzes apagadas e a porta P0 aberta. Nessa situação o alarme não deveria tocar, porém, observa-se que o alarme está tocando, o que caracteriza uma observação anormal do estado atual do sistema. A descrição formal completa dessa situação é dada por: OBS={ $P0 \wedge \neg C \wedge \neg L \wedge A$}.

Detecção de sintomas

A observação OBS inicial do estado do sistema reflete um comportamento anormal com relação ao modelo de comportamento correto do sistema de alarme, indicando a necessidade de se chamar o sistema de diagnóstico.

Geração de hipóteses

Executando-se o algoritmo de Reiter para a seguinte entrada:

$SD \cup COMP \cup OBS$

para a qual temos como resposta o diagnóstico:

Δ={{ $COMP_X$}, {$COMP_W$}, {$COMP_Z$} }

Todo o processo para encontrar o diagnóstico é mostrado na Figura 5.5. Na figura, o primeiro nó (parte a) é rotulado com os componentes da resposta devolvida pelo provador de teoremas que tem como entrada $SD \cup COMP \cup OBS$, que neste exemplo é

{$COMP_X$, $COMP_Y$, $COMP_W$, $COMP_Z$, $COMP_V$}.

De acordo com o passo 2a do algoritmo de Reiter, criamos um s_arco saindo de n para cada elemento do rótulo do nó, isto é, para cada elemento do conjunto

{$COMP_X$,$COMP_Y$,$COMP_W$,$COMP_Z$,$COMP_V$}

O primeiro s_arco (da esquerda para direita), é então ligado a um nó que será rotulado com @ pelo passo 2f do algoritmo, uma vez que o provador de teoremas devolve um conjunto vazio de componentes quando considera o componente $COMP_X$ falho. Para o s_arco seguinte o provador de teoremas devolve os componentes

{$COMP_X$,$COMP_W$,$COMP_Z$,$COMP_V$}

e, de acordo com o passo 2.f.i, devemos mudar o rótulo do nó raiz para

{$COMP_X$,$COMP_W$,$COMP_Z$,$COMP_V$}

e remover todos os arcos partindo de $COMP_Y$ (Figura 5.5 b)).

A Figura 5.5 c), mostra que para o s_arco $COMP_W$ e $COMP_Z$ são criados nós rotulados com @ e para o s_arco $COMP_V$ é criado um nó rotulado por

{$COMP_X$, $COMP_Y$, $COMP_W$, $COMP_Z$}

de acordo com a resposta do provador de teoremas, supondo falho o componente $COMP_V$. Para o nível seguinte, criamos um arco $COMP_X$ que será podado, uma vez que gera uma suposição de falha já analisada anteriormente (de acordo com o passo 2.d do algoritmo). Para $COMP_Y$ criamos o nó

{$COMP_X$,$COMP_W$,$COMP_Z$} (passo 2.f.i).

Finalmente, na parte d) da figura, mudamos o rótulo da raiz para

{$COMP_X$,$COMP_W$,$COMP_Z$}

e removemos todos os nós gerados a partir do s_arco $COMP_V$ que foi eliminado da raiz.

Como não existe mais nós para expandir, já temos o diagnóstico baseado no teorema do algoritmo de Reiter:

Δ={{ $COMP_X$}, {$COMP_W$} , {$COMP_Z$} }

Discriminação de hipóteses

Na tarefa de discriminação de hipótese, selecionamos a primeira hipótese encontrada, nesse caso, o componente $COMP_X$. É feita então uma medida direta do valor da variável de saída de $COMP_X$ (conexão I), observando um valor igual a 0. Aplicando o método de geração de hipóteses baseado em modelos para explicar a nova observação juntamente com as observações iniciais, isto é:

$$OBS' = \{P0 \wedge \neg C \wedge \neg L \wedge A \wedge \neg I\}$$

obtemos o diagnóstico:

Δ'={{$COMP_W$} , {$COMP_Z$} }

Em seguida, selecionamos uma nova hipótese $COMP_W$ e observamos o valor 1 na conexão J. O conjunto das novas observações é:

$$OBS'' = \{P0 \wedge \neg C \wedge \neg L \wedge A \wedge \neg I \wedge J\}$$

que ao ser usado numa nova chamada ao sistema de diagnóstico devolve:

Δ''={{$COMP_W$}}

que é a resposta procurada.

5.5 Sistemas de Diagnóstico Baseado em Modelos da literatura

General Diagnosis Engine (GDE) [dW87] propôs uma *máquina de diagnóstico geral*, que pode detectar falhas únicas e múltiplas (mais difíceis de serem detectadas). O processo de diagnóstico em GDE consiste de três fases: reconhecimento de conflitos, geração de candidatos e discriminação de candidatos.

GDE usa propagação de restrições para calcular os valores de saída. As suposições nas quais esses valores estão baseados são guardadas em registros de dependência. Se diferentes valores para o mesmo ponto são calculados, os conjuntos conflito são obtidos a partir das dependências. Em seguida, os conflitos minimais são transformados em candidatos minimais, conjunto que inclui pelo menos um componente de todos os conflitos,

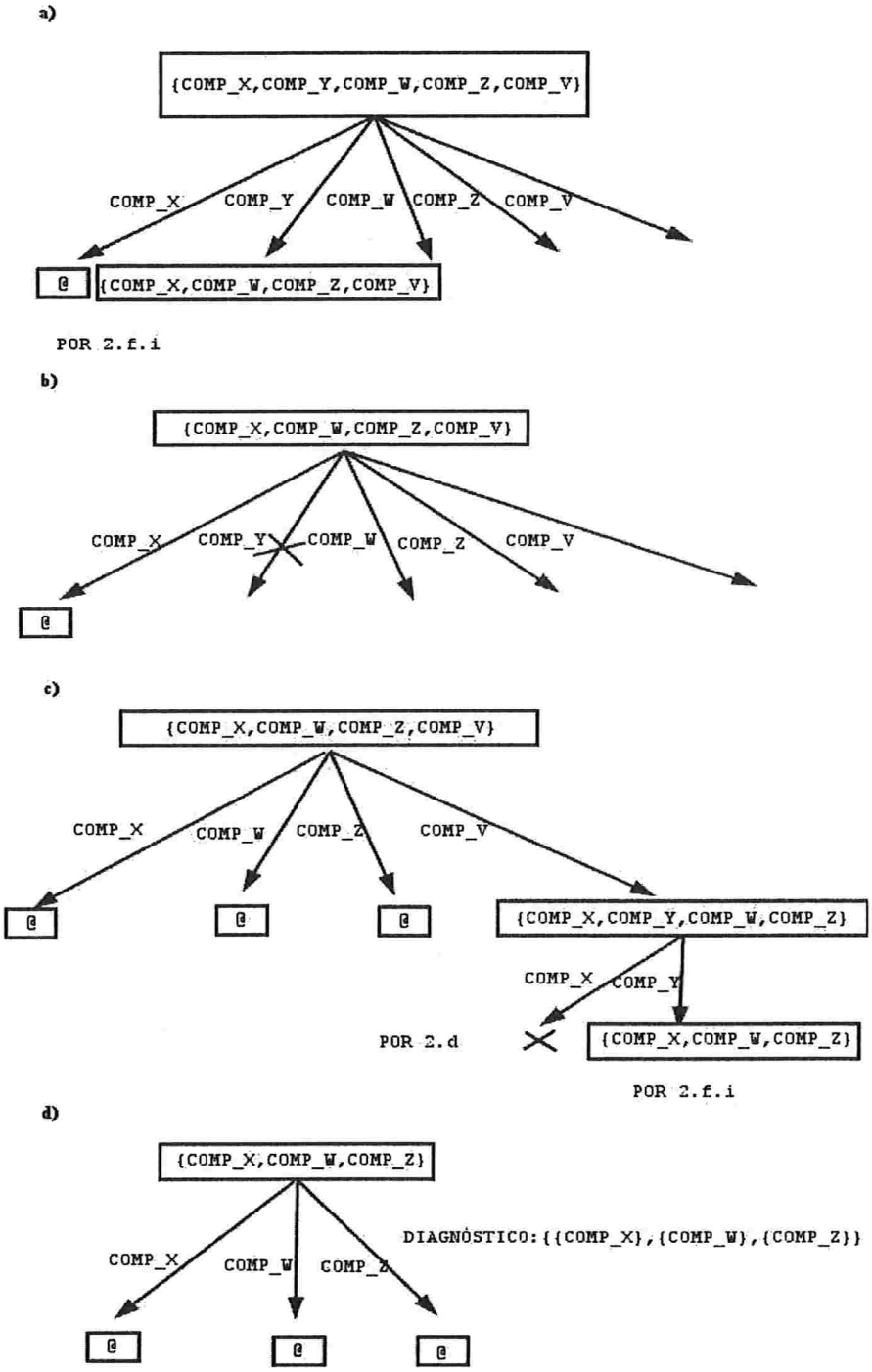


Figura 5.5: Aplicação do algoritmo de Reiter para o circuito de um alarme de carro

que corresponde à construção da árvore de conjuntos de corte feita pelo algoritmo de Reiter.

O sistema de diagnóstico *HT* [Dav84] usa a técnica de *suspensão de restrições*, onde o diagnóstico é o componente cuja suspensão das regras de comportamento (restrições) torna a descrição do sistema consistente com as observações. O algoritmo HT também usa propagação de restrições, faz a suposição de falha única e permite modelar o sistema hierarquicamente.

As definições de diagnóstico baseado em modelos dadas na Seção 5.3 se baseiam no trabalho de Richard Benjamins, entitulado *Problem Solving Methods for Diagnosis* [Ben93], que faz uma avaliação conceitual da tarefa, generalizando termos e conceitos. Por exemplo, no GDE o termo conjunto conflito é usado ao invés de conjunto de contribuintes e o termo candidato ao invés de hipóteses de diagnóstico.

5.6 Diagnóstico hierárquico

Uma das formas de tornar a tarefa de diagnóstico mais tratável é usar o conceito de abstração. As abordagens propostas são tipicamente hierárquicas, tratando o problema em múltiples níveis de detalhe. Falhas são detectadas em diferentes níveis, do mais abstrato ao menos abstrato, reduzindo-se assim o custo computacional do diagnóstico.

Duas classes de abstrações são comumente empregadas: (i) *abstração estrutural*, que agrega componentes para descrever o sistema em diferentes níveis de detalhe estrutural e (ii) *abstração comportamental*, que tenta encontrar uma falha primeiro fazendo uma simplificação no modelo comportamental dos componentes, para depois considerar o modelo completo porém, para uma parte reduzida do sistema [CR04].

5.7 Considerações finais

Nesse capítulo foi apresentado o processo geral de diagnóstico, que consiste em inferir explicações para um conjunto de observações. As sub-tarefas do processo de diagnóstico baseado em modelos: detecção de sintomas, geração de hipóteses e discriminação de hipóteses foram descritas em detalhes. Também foi mostrado um exemplo de aplicação de diagnóstico para um circuito elétrico. No próximo capítulo será especificado o sistema PROPAT_DEBUG que usa a técnica de DBM adaptada para diagnosticar programas.

CAPÍTULO 6

Diagnóstico Baseado em Modelos de Programas

Enquanto que um engenheiro, para determinar as partes falhas de sistemas mecânicos ou eletrônicos, raciocina sobre as diferenças entre o sistema real e o seu modelo de projeto (modelo correto), um programador tenta encontrar falhas em programas raciocinando sobre as diferenças entre o seu código e as suas intenções para satisfazer os objetivos do problema. Nesse capítulo mostramos como as técnicas de Diagnóstico Baseada em Modelos, definidas no capítulo anterior, podem ser aplicadas no diagnóstico de programas.

6.1 Projeto Jade: modelos de programas

Dado que na depuração automática **o programa** é visto como o sistema a ser diagnosticado, uma aplicação original de técnicas de Diagnóstico Baseada em Modelos foi proposta no projeto JADE [MS+02]. A proposta do projeto foi o desenvolvimento de um sistema de depuração automática para ajudar programadores avançados a encontrar seus erros. A idéia básica é: derivar o modelo estrutural diretamente do programa e derivar o modelo comportamental da semântica da linguagem.

A Figura 6.1 mostra um modelo conceitual da tarefa de diagnóstico baseado em modelos de programas (parte b) e faz um paralelo com o diagnóstico de sistemas físicos (parte a) proposto em [Ben93]. Um resumo desse paralelo é feito na Figura 6.2 sendo que as principais mudanças de paradigma são:

- Em problemas de diagnóstico de sistemas físicos o modelo está correto e as observações feitas diretamente no sistema físico refletem o comportamento incorreto (Figura 6.1, parte a). No diagnóstico de programas, o modelo do sistema passa a ser o **modelo do programa que contém erros de programação**; e as observações são as saídas corretamente produzidas pelo programa com falhas (Figura 6.1, parte b).

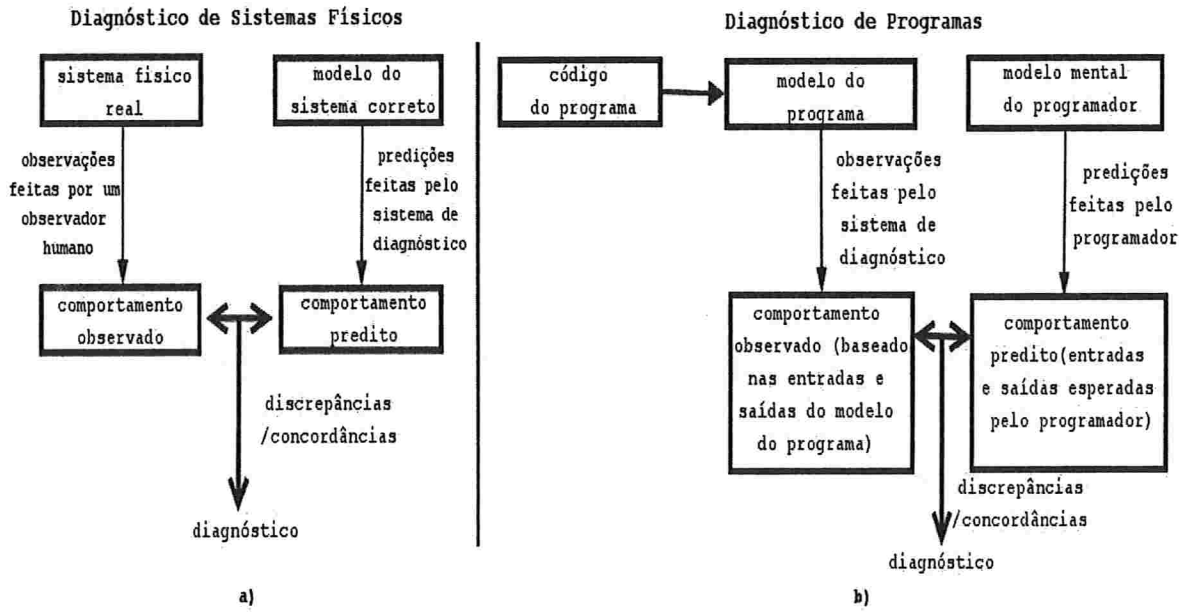


Figura 6.1: Diagnóstico Baseado em Modelos (DBM) visto como a interação de observações e predições. A parte (a) descreve um modelo conceitual de DBM para sistemas físicos e a parte (b) descreve o DBM de programas.

- As predições feitas no diagnóstico de sistemas físicos são feitas pelo sistema de diagnóstico sobre o modelo correto (Figura 6.1, parte a). Já no diagnóstico de programas as predições são feitas pelo programador (Figura 6.1, parte b). Nessa situação, se espera que um programador seja capaz de comunicar suas intenções de programação, que podem ser os valores esperados para as variáveis de acordo com a estratégia empregada pelo programador (assim como a aplicação de um *design pattern* ou de um padrão elementar) para satisfazer as metas e submetas do problema.

No processo de diagnóstico proposto por Mateis *et al.* [MSWW00], o programa é compilado em uma representação interna e, junto com as descrições lógicas das instruções da linguagem, é convertido em um modelo formal. Assim, a construção de modelos é feita automaticamente. O modelo e o comportamento especificado do programa para casos de teste são usados pelo mecanismo de diagnóstico geral para gerar as hipóteses de erro (Figura 6.3).

No Projeto JADE foram feitas duas propostas de modelagem de programas: *modelo baseado em dependência* [Wie01] e *modelo baseado em valor* [MSW00]. Ambas usam a abordagem hierárquica para diagnosticar programas feitos em Java e são usadas para detectar erros no código fonte que se manifestam como um erro

	Diagnóstico Baseado em Modelos de sistemas físicos	Diagnostico Baseado em Modelos de programas
Modelo do Sistema (SD)	modelo do sistema correto	modelo do programa que contém erros de programação
Observaçoes	reflete o comportamento incorreto do sistema	saídas corretamente produzidas pelo programa com falhas
Prediçoes	predições feitas pelo sistema de diagnóstico sobre o modelo correto	predições feitas pelo programador sobre o seu modelo (mental) de solução

Figura 6.2: Principais diferenças entre o DBM de sistemas físicos e o DBM de programas.

de saída (uma observação ou um sintoma anormal). Essas duas propostas são usadas para detectar *falhas funcionais* mas não são recomendadas para encontrar *falhas estruturais* em programas [Wie01] (para as quais a técnica de verificação com respeito à especificação funciona melhor).

Nesse trabalho, foi usado o Modelo Baseado em Valor (Seção 6.3) para o diagnóstico do programa do aluno. Dado que o Modelo Baseado em Valor pode eliminar diagnósticos errados usando informação adicional em tempo de execução (os valores das variáveis), este pode obter melhores resultados que o Modelo Baseado em Dependências, na maioria dos casos [CW03].

6.2 MBD de programas para programadores aprendizes

Na correção de programas feita por um professor de computação, ao invés do professor comparar o programa do aluno com um programa correto, este raciocina sobre a semântica de cada expressão ou sentença do programa do aluno. O professor pode também raciocinar em termos de estratégias gerais de programação, tentando identificá-las dentro do código do estudante e analisar se estas estão bem implementadas para resolver o problema atual. Por exemplo, se para um problema que envolve o tratamento de uma seqüência de elementos não existir um comando de repetição no programa do aluno, o professor imediatamente identificaria um erro. Se o professor não reconhecer nenhum erro estrutural evidente ou não identificar o uso de uma estratégia geral, o professor deve analisar linha a linha o programa, quase como numa simulação.

A técnica de MBD de programas se assemelha à correção feita pelo professor através da simulação. A única diferença ao aplicar essa técnica para alunos, ao invés de programadores experientes, está na forma como as hipóteses deverão ser comunicadas ao aluno.

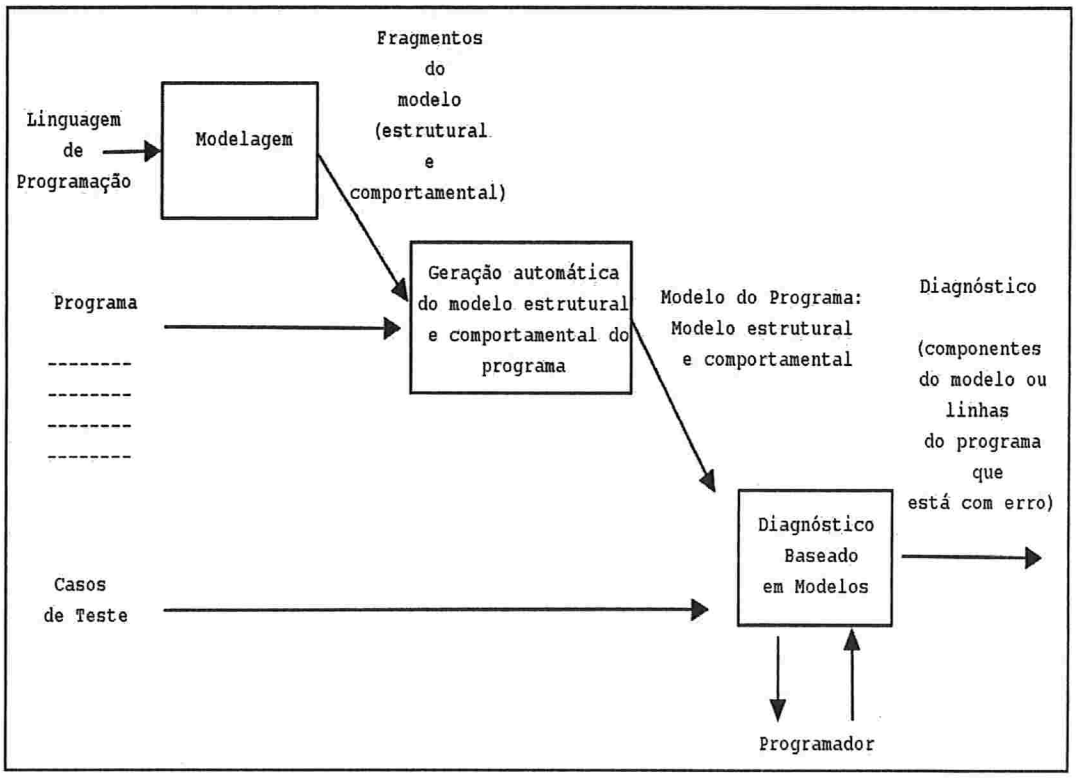


Figura 6.3: Processo de Diagnóstico Baseado em Modelos do projeto JADE

6.3 O Modelo Baseado em Valor

No diagnóstico de programas, o modelo baseado em valor define a modelagem estrutural e comportamental do programa que corresponde à descrição do sistema, SD, no diagnóstico de sistemas físicos (Capítulo 5). Nesse modelo, expressões e sentenças são representadas como componentes, sendo a semântica das expressões e das sentenças descritas por um conjunto de fórmulas da lógica (*fragmentos do modelo* da Figura 6.3). Nesse modelo, os componentes estão conectados se houver um fluxo de informação entre eles. O fluxo de informação ocorre entre dois componentes, por exemplo, entre uma atribuição e uma seleção simples, quando a atribuição muda a variável que é usada na seleção e não há entre esses dois componentes outra atribuição que mude essa variável [MSW00].

Para obter o modelo estrutural de um programa, Java Mateis *et al.* [MSW00] fazem as seguintes conversões:

Statements	::=	StatementList
StatementsList	::=	Statement StatementList'| $\emptyset$
Statement	::=	Assigment | SelectionStatement | IterationStatement | FunctionCallVoid | Read | Print
Assignment	::=	id=Expr;
Selection	::=	*if* (Expr) $Statements_{then}$[*else* $Statements_{else}$]
IterationStatement	::=	*while* (Expr) *Statements*
Expr	::=	id | const | $Expr_1$ *Op* $Expr_2$ | FunctionCall
FuctionCall	::=	id() | id (ParList)
FunctionCallVoid	::=	id() | id (ParList)
ParList	::=	Expr | Expr, ParList'

Tabela 6.1: Gramática BNF da linguagem C restrita. Em que Op é um operador binário que pode ser substituído por: $+$, $-$, $*$, $/$, $\%$, $<$, $>$, $<=$, $>=$, $==$, $!=$, $\&\&$ ou $\|$ e *const* e *id* podem ser do tipo inteiro ou booleano.

- a cada componente estão associadas n portas de entrada e m portas de saída. Portas de entrada correspondem a variáveis usadas pelo componente. Portas de saída correspondem a variáveis modificadas pelo componente. Algumas portas de entrada e de saída são ainda criadas no modelo para representar resultados intermediários de fragmentos de programas, por exemplo, na avaliação de expressões;
- todas as sentenças: atribuições, condicionais, laços *while*, sentenças *return* e chamadas a funções, são mapeadas a componentes;
- todas as variáveis são mapeadas para conexões e sempre que uma variável ocorre em uma expressão ou sentença (componente), esta conexão é mapeada a uma entrada (se seu valor for usado) e/ou saída (se seu valor for modificado) do componente.

O modelo comportamental do programa é construído a partir da descrição de cada comando com base na semântica definida pela linguagem de programação, no nosso caso, a linguagem C. A linguagem usada para descrever o modelo comportamental é baseada na Lógica de Predicados de Primeira Ordem, com os seguintes predicados especiais:

- o predicado $AB(C)$ é usado para indicar que o componente C tem um comportamento anormal [1]. Assim, um componente C com funcionamento correto é indicado por $\neg AB(C)$.
- a porta de entrada k de um componente C é representada pelo predicado $in_k(C)$ e o conjunto de todas as entradas de C é representado pelo predicado $INPUT(C)$. Se C' é uma parte do componente C, por

[1] Um componente é considerado anormal se os valores de saída esperados pelo programador são diferentes dos valores calculados pelo componente.

exemplo, a parte *else* do componente de seleção alternativa, chamamos de $INPUT(C')$ ao conjunto de todas as entradas de C'.

- a saída k de um componente C é representada pelo predicado $out_k(C)$ e o conjunto de todas as saídas de C é representado pelo predicado $OUTPUT(C)$. Se C' é uma parte do componente C, por exemplo, a parte *else* do componente de seleção alternativa, chamamos de $OUTPUT(C')$ ao conjunto de todas as saídas de C'. As saídas que são resultado de expressões são denominadas de $result(C)$;
- se um componente C só possui uma porta de entrada ou de saída, esta será representada somente por $in(C)$ ou $out(C)$;
- para um componente condicional C (por exemplo, um comando de seleção ou de repetição) a expressão condicional é transformada num componente independente cuja porta de saída está conectada a uma porta de entrada de C. Representamos essa porta de entrada especial através do predicado $condResult(C)$.

Nas próximas seções serão apresentadas as conversões definidas em [MSW00] para obter o modelo estrutural e comportamental, a partir de um programa. Nesse trabalho, foi considerado apenas um subconjunto da linguagem C, que chamaremos de *linguagem C restrita*, definida na Tabela 6.1 pela gramática BNF. A linguagem C restrita contém grande parte da linguagem usada num curso introdutório de programação sendo possível estendê-la em trabalhos futuros.

6.3.1 EXPRESSÕES

Na tabela 6.1 expressões (Expr), da gramática BNF são construções usadas em sentenças mas que não são sentenças, com exceção das chamadas a funções do tipo *Void* (FunctionCallVoid). Temos quatro tipos fundamentais de expressões: variáveis (id), constantes (const), expressões com operadores ($Expr_1$ Op $Expr_2$) e chamadas a funções (FunctionCall). Expressões são modeladas como componentes (Figura 6.4), com exceção de variáveis e constantes que são modeladas como conexões.

EXPRESSÕES: OPERADORES BINÁRIOS

BNF: $Expr ::= Expr_1\ Op\ Expr_2$

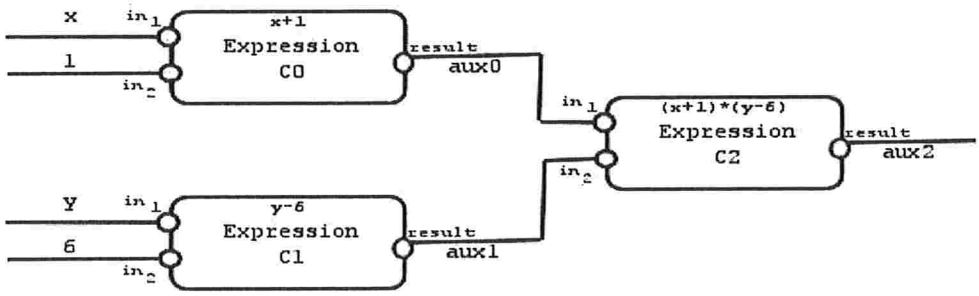
- **Modelo Estrutural**: operadores binários são mapeados a componentes com duas portas de entrada (in_1 e in_2) e uma porta de saída (*result*). Por exemplo, na Figura 6.4 a operação aritmética $(x+1)*(y-6)$ é mapeada nos componentes $C0$, $C1$ e $C2$, todos do tipo *Expression*.

  A partir da definição recursiva da BNF, as entradas de uma expressão podem ser o resultado de outras duas expressões (Figura 6.4). No exemplo, as portas de entrada $in_1(C)$ e $in_2(C)$ são conectadas às portas de saída $result(Expr_1)$ e $result(Expr_2)$ dos componentes correspondentes a $Expr_1$ e $Expr_2$. E a porta

Fragmento de programa (Expressão)

(x+1)*(y-6)

Modelo estrutural



in$_1$(C0)=x

in$_2$(C0)=1

result(C0)=aux0

in$_1$(C1)=y

in$_2$(C1)=6

result(C1)=aux1

in$_1$(C2)=aux0

in$_2$(C2)=aux1

result(C2)=aux2

Modelo comportamental

(1)-AB(C0)-> result(C0)=in$_1$(C0)+ in$_2$ (C0)

(2)-AB(C1)-> result(C1)=in$_1$(C1)- in$_2$ (C1)

(3)-AB(C2)-> result(C2)=in$_1$(C2)* in$_2$ (C2)

Figura 6.4: Modelo de estrutural e comportamental para uma **expressão**.

de saída $result(C)$ é associada à variável auxiliar cujo valor é o resultado da avaliação do operador Op com os operandos $result(Expr_1)$ e $result(Expr_2)$. Na Figura 6.4, no modelo estrutural, são declarados todos os mapeamentos entre conexões e variáveis para o fragmento de programa do exemplo.

- **Modelo Comportamental**: Dado C um componente do tipo *Expression*, temos:

$$\neg AB(C) \rightarrow result(C) = in_1(C)\ Op\ in_2(C)$$

No modelo comportamental na Figura 6.4, o componente $C2$ corresponde à expressão $(x+1)*(y-6)$ e suas entradas correspondem às avaliações (*result*) dos componentes $C0$ e $C1$, sendo que: $C0$ modela a expressão $(x+1)$ e $C1$ modela a expressão $(y-6)$. Para declarar o comportamento normal do componente $C2$ dizemos:

$$\neg AB(C2) \rightarrow result(C2) = in_1(C2) * in_2(C2)$$

EXPRESSÕES: CONSTANTES e VARIÁVEIS

Constantes e variáveis são mapeadas a conexões. Toda vez que uma variável é usada no lado esquerdo de uma atribuição, uma nova conexão é criada e usada para todos os componentes que usam essa variável até que a variável seja novamente usada no lado esquerdo de uma atribuição. No exemplo da Figura 6.5, quando o valor da variável *start* é modificado por uma atribuição, por exemplo, uma nova conexão deve ser criada (*start*1).

6.3.2 ATRIBUIÇÃO

BNF: Assignment::= id=Expr;

Modelo Estrutural

Atribuições (Figura 6.5) são mapeadas a componentes do tipo *Assignment* com uma porta de entrada *in(C)* e uma porta de saída *out(C)*, sendo que:

- a porta de entrada $in(C)$ está relacionada a avaliação da expressão (*Expr*) do lado direito da sentença de atribuição (que pode ser uma operação, uma variável, uma constante ou uma chamada a função). Caso a atribuição seja de uma variável ou constante (isto é, uma conexão), $in(C)$ é mapeada diretamente a essa conexão. Caso a atribuição seja de uma expressão, essa é transformada em um outro componente, cuja porta de saída (*result*) é conectada a $in(C)$;
- a porta de saída $out(C)$ do componente *assignment* está relacionada com a variável do lado esquerdo da sentença de atribuição (*id*).

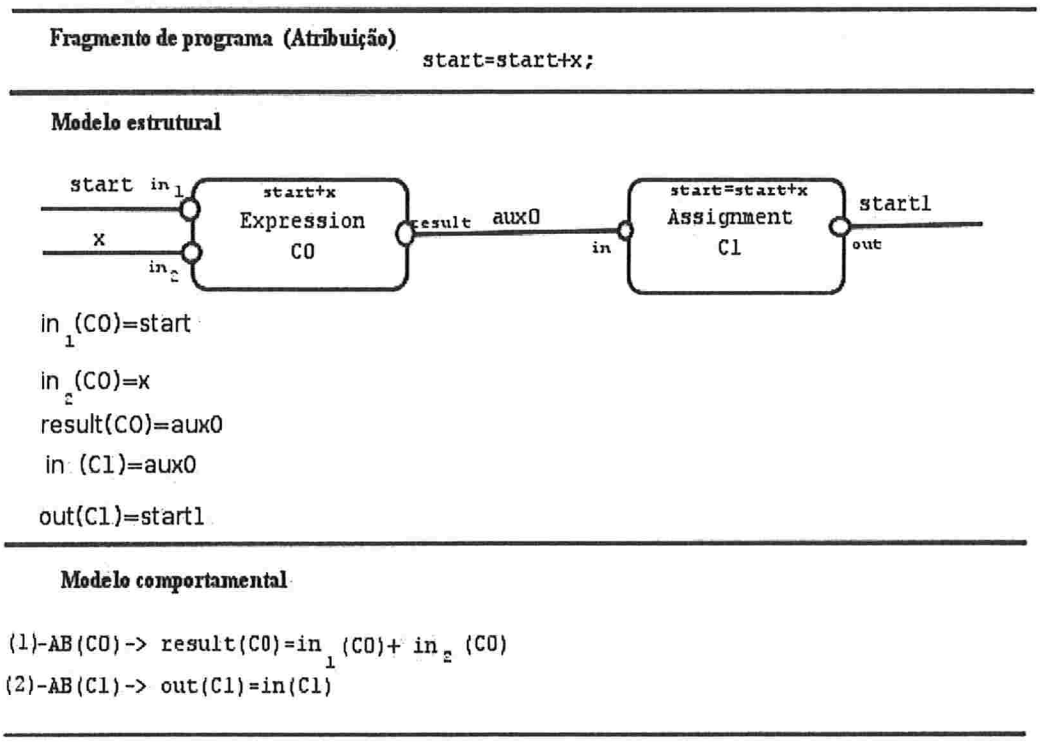


Figura 6.5: Modelo estrutural e comportamental para uma **atribuição**.

Na Figura 6.5 a atribuição $start = start + x$ é mapeada para o componente $C1$ do tipo *Assignment* e o componente $C0$ do tipo *Expression*.

Modelo Comportamental

De uma forma geral, sendo C um componente do tipo *Assigment*, temos:

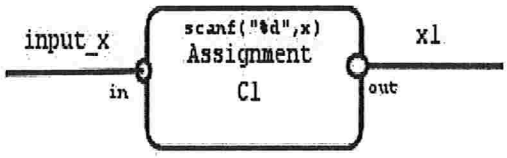
$\neg AB(C) \rightarrow out(C) = in(C)$

Na segunda sentença do modelo comportamental na Figura 6.5, $in(C1)$ é a entrada de $C1$ e $out(C1)$ é a saída. Essa sentença especifica que se o componente $C1$ está normal, $\neg AB(C1)$, implica em $out(C1) = in(C1)$.

Fragmento de programa (scanf)

```
scanf("%d",x);  <=> x1=input_x
```

Modelo estrutural



in (C1)=input_x

out(C1)=x1

Modelo comportamental

-AB(C1)-> out(C1)=in(C1)

Figura 6.6: Modelo estrutural e comportamental para uma sentença de **leitura**.

6.3.3 LEITURA E IMPRESSÃO

As funções da biblioteca de C, *scanf* e *printf*, são tratadas como atribuições. Por exemplo, o comando $scanf(\text{"}\%d\text{"}, \&a)$ é modelado como $a = input_a$. O mesmo foi feito para o comando *printf*.

Para outras funções da biblioteca de C, como por exemplo *sqrt* ou *pow*, é preciso construir modelos mais complexos, ou seja, modelos que envolvam vários componentes da linguagem. Nesse trabalho, não tratamos essas funções, mas elas podem ser facilmente implementadas.

6.3.4 SELEÇÃO

BNF: Selection ::= if (Expr) $Statements_{then}[else\ Statements_{else}]$

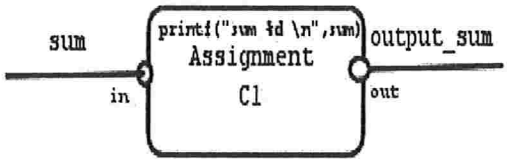
Modelo Estrutural

Seleções são mapeadas em um componente C do tipo *Conditional*, com várias portas de entrada e saída,

Fragmento de programa (printf)

```
printf("sum %d \n",sum);  <=> output sum=sum;
```

Modelo estrutural



in (C1)=sum

out(C1)=output_sum

Modelo comportamental

-AB(C1)-> out(C1)=in(C1)

Figura 6.7: Modelo estrutural e comportamental para uma sentença de **impressão**.

e um componente *Expression* para a expressão condicional. Um componente *Conditional* é composto por:

- uma porta de entrada $condResult(C)$ que é conectada à porta de saída $result$ do componente *Expression* (isto é, a avaliação da expressão condicional);
- um conjunto $INPUT(C)$ de portas de entrada, uma para cada variável usada no ramo *then* ou *else*, conectadas às respectivas variáveis do programa (isto é, às últimas conexões criadas para essas variáveis);
- um conjunto $OUTPUT(C)$ de portas de saída, uma para cada variável que é modificada no ramo *then* ou *else*;
- o subsistema *C_then* contém todos os componentes e conexões do ramo *then* ($Statements_{then}$). Esse subsistema contém também um conjunto de portas de entrada $INPUT(C_then)$, que corresponde ao conjunto $INPUT(C)$ mas que não estão efetivamente conectadas entre si. E um conjunto de portas de saída $OUTPUT(C_then)$, que corresponde ao conjunto $OUTPUT(C)$ mas que também não estão efetivamente conectadas entre si. A idéia é que essas portas só serão efetivamente conectadas, caso o ramo *then* seja executado;

- o subsistema *C_else* contém todos os componentes e conexões do ramo *else* ($Statements_{else}$). Esse subsistema contém também um conjunto de portas de entrada $INPUT(C_else)$, que corresponde ao conjunto $INPUT(C)$ mas que não estão efetivamente conectadas entre si. E um conjunto de portas de saída $OUTPUT(C_else)$, que corresponde ao conjunto $OUTPUT(C)$ mas que também não estão efetivamente conectadas entre si. A idéia é que essas portas só serão efetivamente conectadas, caso o ramo *else* seja executado.

É importante notar que todas as variáveis de saída do componente condicional devem fazer parte dos dois subsistemas, *C_then* e *C_else*, ou seja os conjuntos $OUTPUT(C_then)$ e $OUTPUT(C_else)$ possuem a mesma quantidade de elementos. Dependendo de qual ramo for executado, um desses conjuntos será efetivamente conectado ao conjunto $OUTPUT(C)$.

A Figura 6.8 mostra a modelagem de um fragmento de programa que inclui uma sentença condicional, note que no *thenModel* existe uma conexão para a variável *to* e no *elseModel* existe uma conexão para a variável *from*, ainda que elas não sejam usadas nos subsistemas respectivos, o que permite também a propagação progressiva e regressiva dos valores.

Modelo Comportamental

Para cada variável x que é modificada em pelo menos um dos ramos *then* ou *else*, temos o seguinte modelo comportamental:

$$\neg AB(C) \wedge condResult(C) = true \rightarrow \forall\ k \mid in_k(C) \in INPUT(C),\ in_k(C) = in_k(C_then)$$

$$\neg AB(C) \wedge condResult(C) = false \rightarrow \forall\ k \mid in_k(C) \in INPUT(C),\ in_k(C) = in_k(C_else)$$

$$\neg AB(C) \wedge condResult(C) = true \rightarrow \forall\ k \mid out_k(C) \in OUTPUT(C),\ out_k(C) = out_k(C_then)$$

$$\neg AB(C) \wedge condResult(C) = false \rightarrow \forall\ k \mid in_k(C) \in OUTPUT(C),\ out_k(C) = out_k(C_else)$$

6.3.5 WHILE

BNF: IterationStatement ::= *while* (Expr) *Statements*

O modelo do comando *while*, chamado *While Livre de Laço* [MSWW02], transforma laços em um conjunto de sentenças condicionais aninhadas. O número de sentenças condicionais é calculada a partir do número de iterações que o laço é executado para um dado caso de teste, o que permite determinar falhas das sentenças dentro do laço. Um exemplo de *While Livre de Laço* é dado na Figura 6.9.

Fragmento de programa (Seleção)

```
if(from<to)
  start=from
else
  start=to
```

Modelo estrutural

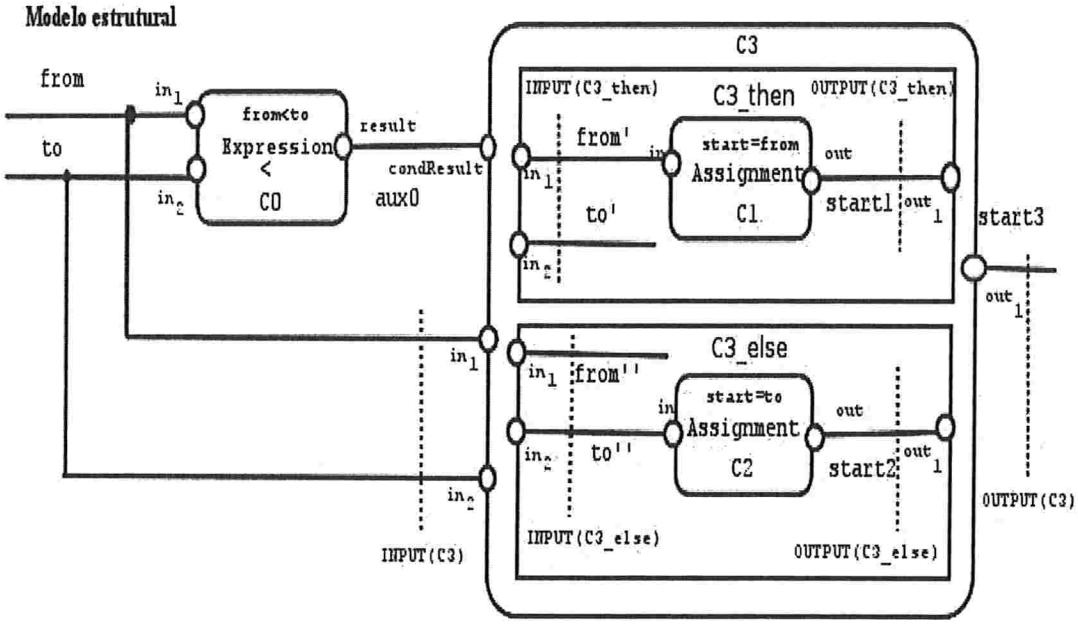


Figura 6.8: No modelo baseado em valor, um comando de **seleção** (*if* ou *if-else*) é mapeado a um componente *conditional* com dois subsistemas *thenModel* e *elseModel*. O modelo comportamental desse exemplo foi omitido por motivos de espaço.

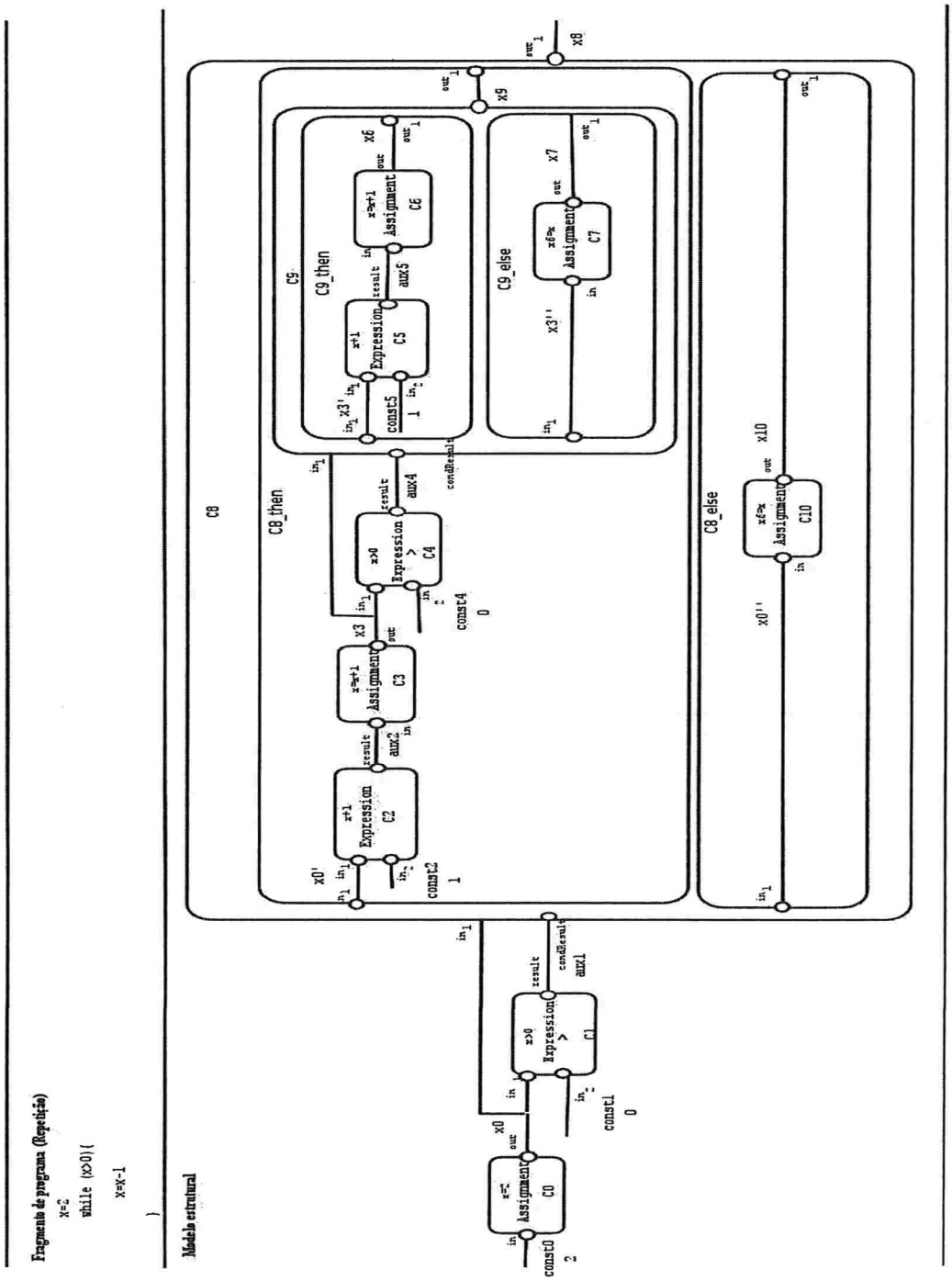


Figura 6.9: Modelo estrutural e comportamental para uma sentença **while**.

6.3.6 FUNÇÕES do tipo VOID

Modelo Estrutural

Chamadas a funções do tipo *void* podem modificar variáveis externas à função através da passagem de parâmetros por referências (endereços) ou variáveis globais. As chamadas a funções são mapeadas como componentes *FunctionCallVoid* com várias portas de entrada e saída:

- as portas de entrada estão relacionadas às variáveis globais usadas no corpo do método chamado e os parâmetros atuais da chamada da função.
- As portas de saída estão relacionadas às variáveis globais usadas no corpo da função e os parâmetros que são modificados dentro do corpo.

Modelo Comportamental

Substituir todos os parâmetros formais e variáveis globais com os correspondentes parâmetros atuais e variáveis globais.

6.3.7 FUNÇÕES que devolvem valores

Modelo Estrutural

Chamadas a funções que devolvem valores são mapeadas como componentes *FunctionCall*, de forma similar aos componentes do tipo *FunctionCallVoid*. Adicionalmente tem-se uma porta de saída *result* relacionada ao valor devolvido pela chamada à função.

Modelo Comportamental

A descrição do comportamento é similar à chamada a métodos, sendo que o resultado da expressão da sentença *return* da função será associado a porta de saída *result*.

6.3.8 ProPAT_deBUG: construção do modelo de componentes e conexões

Para construir o modelo completo de componentes e conexões do programa do aluno, é criada uma árvore sintática abstrata (Abstract Syntac Tree - AST)[2] a partir do programa escrito na linguagem C. Para isso, foi

[2]Árvores sintáticas abstratas (AST) são freqüentemente usadas como a representação interna de compiladores ou interpretadores de um programa. Enquanto uma árvore sintática é um registro das regras da gramática usadas para um

construçãoModelo(programa do aluno)

1. Criar a AST apartir do programa do aluno
2. **Percorrer** o AST para
   (a) criar os componentes
   (b) identificar o conjunto de variáveis que são usadas e modificadas pelos componentes.
3. **Percorrer** o AST para
   (a) criar novas versões para todas as variáveis (conexões) modificadas.
   (b) conetá-las aos componentes correspondentes.

Figura 6.10: Algoritmo de criação do modelo de componentes e conexões do programa do aluno.

usada a ferramenta ANTLR (*ANother Tool for Language Recognition* [Par89]), um arcabouço para construir compiladores e tradutores a partir de descrições gramaticais para Java, C++, Python, entre outras.

A árvore sintática é usada para determinar: (i) a estrutura do programa, (ii) o conjunto de variáveis usadas e (iii) as cadeias de definição e uso das variáveis. A Figura 6.10 mostra o algoritmo geral usado para gerar o modelo de componentes e conexões a partir da AST. No passo 2 a árvore é percorrida com o objetivo de criar componentes e identificar variáveis.

No passo 3 do algoritmo 6.10 são criadas novas versões das variáveis, por exemplo, depois de fazer uma atribuição no programa $start = from + x$, uma nova versão da variável start é criada e é usada por todos os componentes subseqüentes, até que lhe seja atribuído um novo valor onde novamente uma nova versão da variável deve ser criada.

6.4 ProPAT_deBUG: o Diagnóstico Baseado em Modelos

Nesta seção, mostramos como as três tarefas do processo de diagnóstico (Figura 6.11): detecção de sintomas, geração de hipóteses e discriminação de hipóteses foram implementadas para o diagnóstico de programas.

A Figura 6.12 descreve o algoritmo completo de diagnóstico implementado no sistema ProPAT_deBUG na linguagem Java, que recebe como entrada o programa do aluno e um arquivo contendo os casos de teste. A saída do sistema é o diagnóstico. Os passos do algoritmo são descritos nas próximas seções.

programa, a árvore sintática abstrata registra a estrutura do programa mas é independente da gramática da linguagem.

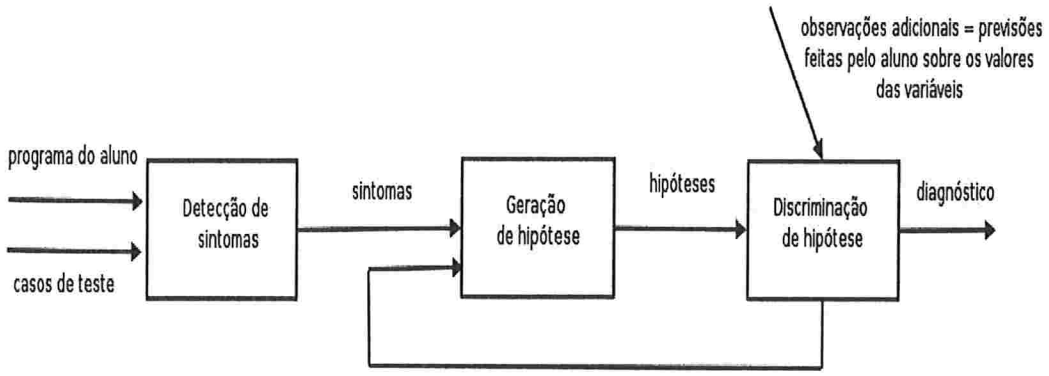


Figura 6.11: Processo de diagnóstico do sistema **ProPAT_deBUG**

6.4.1 Detecção de sintomas do programa

Dado um conjunto de casos de teste, representados como uma tabela de valores de entrada e saída corretos para a solução do problema, um sintoma é qualquer diferença entre as saídas do programa do aluno e as saídas do caso de teste. A tarefa de diagnóstico só será executada se algum sintoma for detectado para algum caso de teste.

Note que, ao invés de termos uma tabela de entradas e saídas, poderiamos adicionar para cada problema, um programa solução dado pelo professor. Assim, bastariam as entradas (dadas pelo professor ou geradas aleatoriamente) para gerar as saídas executando-se o programa para as entradas.

6.4.2 Geração de hipóteses do programa

Como visto no Capítulo 5, essa tarefa pode ser decomposta em três sub-tarefas: *busca dos contribuintes*, *transformação dos contribuintes em um conjunto hipóteses* e *predição baseada em filtragem.*

Busca dos contribuintes

O conjunto de contribuintes é um conjunto de componentes que contém pelo menos um componente falho (Definição 5.4). Para determinar os conjuntos de contribuintes, ao invés de se usar um sistema do tipo ATMS junto com um provador de teoremas, como descrito no Capítulo 5, nesse trabalho foi implementado um sistema de regras de produção [RN03] com registro de dependências. Assim, o modelo comportamental é representado por um conjunto de regras de produção, que podem ser encadeadas para propagar, no caso de regras de programas, valores de variáveis pelas conexões e portas de componentes. As regras (modelo comportamental)

diagnóstico(programa do aluno, casos de teste)

1. Detectar sintomas.
2. **Se** algum sintoma é detectado **então**:
   (a) Converter o programa do aluno no modelo de componentes e conexões.
   (b) **Para cada** caso de teste
      i. Classificar o caso de teste como bem sucedido ou falho.
      ii. Calcular o valor estimado de falha de cada componente.
   (c) **Para** o caso de teste com menor número de iterações que gerou saídas com erro **fazer**:
      i. Propagar os valores pelas conexões e portas de componentes, usando regras de simulação e de inferência, armazenando as dependências a fim de encontrar uma inconsistência (valores diferentes numa mesma conexão).
      ii. Buscar os contribuintes com base nas dependências armazenadas.
      iii. Transformar os contribuintes em um conjunto de hipóteses usando o algoritmo de Reiter.
      iv. Descartar hipóteses que incluem componentes de leitura e escrita.
      v. Mostrar ao aluno as hipóteses mais prováveis, de acordo com o valor estimado calculado.
      vi. **Para cada** hipótese na ordem de execução no programa faça:
         A. Perguntar ao aluno os valores esperados para as variáveis de saída dos componentes que pertencem a hipótese.
         B. Executar novamente o sistema de diagnóstico para o programa do aluno, o caso de teste e os novos valores dados (previstos) pelo aluno.
         C. **Se** o conjunto de hipóteses for unitário **Então exit**
3. **Senão** (nenhum sintoma detectado) a tarefa de diagnóstico não será executada.

Figura 6.12: Algoritmo de diagnóstico do sistema PROPAT_DEBUG

(a) SE o valor de um dos operandos é igual a 0 ENTÃO o resultado é 0.

(b) SE os valores dos operandos são conhecidos ENTÃO o valor da saída é igual ao primeiro operando vezes o segundo operando.

(c) SE o valor de um dos operandos e o valor da saída são conhecidos ENTÃO o valor do outro operando é igual à divisão desses valores. Se o resultado da divisão não é inteiro então existe um erro.

(d) SE o valor da saída é igual a 1 ENTÃO os valores dos operandos são iguais a 1.

Figura 6.13: Regras de produção para o componente multiplicação entre inteiros. Essas regras podem ser usadas para propagar valores para frente ou para trás no modelo do programa.

foram construídas de tal forma que se um subconjunto de valores de entrada e saída são conhecidos, novos valores são calculados e propagados. Por exemplo, o comportamento da operação de multiplicação entre dois inteiros pode ser especificado com as regras da Figura 6.13.

Como definido na Seção(5.1), um sistema pode ser descrito através de dois tipos de regras: *regras de simulação* e *regras de inferência* [Ben93]. Na Figura 6.13 (a) e (b) são regras de simulação, enquanto (c) e (d) são regras de inferência. As regras de simulação propagam os valores da entrada para a saída dos componentes. As regras de inferência propagam os valores da saída para a entrada dos componentes. Nos modelos dos comandos da linguagem C restrita apresentados na Seção 6.3, foram dadas apenas as regras de simulação dos modelos comportamentais. As regras de inferência foram omitidas no texto mas foram usadas no sistema PROPAT_DEBUG.

Para determinar inconsistências no modelo, são executados dois tipos de propagação: a *propagação progressiva* (usando regras de simulação) e a *propagação regressiva* (usando regras de inferência). Uma discrepância ocorre quando o processo de encadeamento de regras gera valores diferentes para uma mesma conexão. Neste caso, a inconsistência é detectada.

Para determinar o conjunto de contribuintes envolvidos em uma inconsistência, foi implementada uma estratégia do tipo ATMS (Capítulo 5): enquanto os valores das variáveis são propagados através das regras de produção, é feito um registro dos componentes que derivaram aquele valor e quais valores de entrada foram usados (*justificativa* para os valores calculados). Esse registro é então usado para achar os contribuintes. Isso leva a um grafo dirigido que conecta valores propagados com as restrições e outros valores do qual depende [dW87].

Transformação dos contribuintes num conjunto de hipóteses

Os conjuntos de contribuintes são transformados em um conjunto de hipóteses através do algoritmo de Reiter (Seção 5.3.2). Cada elemento do conjunto de hipóteses é uma explicação de todas as observações iniciais [Ben93] e é composto por um conjunto de componentes do sistema, supostamente falhos.

Predição baseada em filtragem

Usando o conhecimento existente sobre o modelo do sistema, algumas hipóteses são excluídas, por exemplo, no caso de programas, assume-se que os componentes de leitura e impressão estão sempre corretos.

6.4.3 Discriminação de hipóteses do programa

Seleção da hipótese

Visto que o sistema de diagnóstico para programas devolve várias hipóteses, é preciso selecioná-las em uma determinada ordem, a fim de comunicá-la ao aluno e ajudá-lo a encontrar seus erros. Para isso, pode-se escolher uma das seguintes estratégias:

1. selecionar hipóteses de maneira aleatória;
2. selecionar hipóteses na ordem em que foram geradas pelo sistema de diagnóstico;
3. selecionar hipóteses na ordem de dificuldade (por exemplo, expressões mais simples antes das mais complexas);
4. selecionar primeiro as hipóteses que incluem maior quantidade de componentes falhos;
5. selecionar primeiro as hipóteses que incluem menor quantidade de componentes falhos;
6. selecionar a hipótese de acordo com a ordem de execução do programa do aluno;
7. ordenar as hipóteses de acordo com o valor estimado das sentenças estarem corretas (cálculo descrito a seguir);
8. ordenar as hipóteses pela quantidade de componentes falhos e como segundo critério de ordenação usar o valor estimado das sentenças estarem corretas.

Dentre as estratégias de discriminação de hipóteses descritas acima, foram implementadas a 6 e a 8. A estratégia 8 serve para apresentar as hipóteses ao aluno numa ordem de menor estimativa dos componentes estarem corretos (ou a maior estimativa de estarem falhos).

Estimativa das sentenças estarem corretas

No sistema PROPAT_DEBUG, o conjunto de hipóteses a serem discriminadas foi ordenado segundo a heurística proposta em [JHS02], detalhada a seguir.

Dado um conjunto de casos de teste T para um programa P e um caso de teste $t \in T$, precisamos de dois tipos de informação relacionadas à execução de P com t: (1) se o caso de teste foi *bem sucedido* ou *falhou* e (2) as sentenças do programa que são executadas quando P é executado com t. O caso de teste foi bem sucedido com t se a saída atual para a execução de P com t é a mesma que a saída esperada por t, caso contrário, o caso de teste falhou. Para cada sentença $s \in P$, é encontrada a estimativa $h(s)$ da sentença estar correta [JHS02], baseada em porcentagens de sucesso e falha.

$$h(s) = \frac{\%sucesso(s)}{\%sucesso(s) + \%falha(s)} \tag{6.1}$$

$$\%sucesso(s) = \frac{numsucs}{totalsuc} \tag{6.2}$$

$$\%falha(s) = \frac{numfalhs}{totalfalh} \tag{6.3}$$

Em que:

numsucs= número de casos de teste bem sucedidos que executaram s

numfalhs=número de casos de teste falhos que executaram s

totalsuc=número total de casos de teste bem sucedidos no conjunto de casos de teste

totalfalh=número total de casos de teste falhos no conjunto de casos de teste

Por exemplo, a estimativa $h(s)$ para a linha 7 no programa da Figura 6.14 é:

$h(s_7) = \frac{\%sucesso(s_7)}{\%sucesso(s_7)+\%falha(s_7)} = \frac{1/5}{1/5+1/1} = \frac{1}{6} = 0.1666$

Segundo o calculo das estimativas baseadas no casos de teste para o programa, a sentença 7 tem o menor valor estimado (0.1666) de estar correta.

A estimativa para duas ou mais sentenças do programa estarem falhas (caso de múltiplas falhas) é dada pela soma dos valores estimados, que chamaremos aqui de *estimativa da hipótese*:

$$h(Hipotese) = \sum_{s \in Hipotese} h(s)$$

mid(){ int x,y,z,m;	cobertura (sentenças executadas pelo caso de teste para x y e z) 1,1,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	estimativa h(s)
1 scanf("%d %d %d",&x,&y,&z);	x	x	x	x	x	x	0,5
2 m=z;	x	x	x	x	x	x	0,5
3 if (y<z)	x	x	x	x	x	x	0,5
4 if (x<y)		x					1
5 m=y;		x					1
6 else if (x<z)	x				x	x	0.288
7 m=y;	x					x	0,16
8 else	x		x	x			1
9 if (x>y)			x				1
10 m=y;			x				1
11 else if (x>z)							ND
12 m=x;							ND
13 printf("middle number is: ", m);	x	x	x	x	x	x	0.5
}							
caso de teste bem sucedido	S	S	S	S	S	N	

Figura 6.14: Estimativa das sentenças estarem corretas, baseadas em casos de teste, para o programa que imprime a mediana de 3 valores x, y e z [JHS02]

Coleta de dados

Após a seleção da primeira hipótese de falha a ser verificada, é preciso interagir com o aluno para coletar suas predições com respeito aos valores das variáveis em seu programa, isto é, das intenções do aluno. Isso deve ser feito de forma que ele possa aprender durante esse processo, ou seja, aprender a programar durante a detecção e correção de seus erros.

Como foi visto na definição de diagnóstico de programas (Seção 6.1), enquanto o sistema propaga e observa os valores de variáveis nas conexões do modelo, o aluno é consultado para fazer uma predição desses mesmos valores. Apesar do projeto JADE [MS+02] ter alcançado alguns resultados interessantes, acreditamos que a maneira como as hipóteses são comunicadas a um programador não seja adequada a um aluno de Introdução à Computação. Enquanto que no projeto JADE espera-se que o programador seja capaz de corrigir seus erros simplesmente olhando as partes falhas com maior estimativa de falha em seu programa, um estudante achará

esta tarefa muito difícil e provavelmente não será capaz de usar o sistema de diagnóstico para aprender a programar.

Para pedir que o aluno faça uma predição de valores nas conexões de saída dos componentes que participam da hipótese a ser discriminada, são usados *scripts* de diálogos para cada sentença da linguagem com base nos Padrões Elementares (Seção 4). Com isso, espera-se que o aluno que aprenda e programe usando os Padrões Elementares no ProPAT, seja capaz de entender melhor a comunicação das hipóteses de falha das pelo sistema de diagnostico. Um exemplo de um *script* usado para discriminação de hipóteses baseado em um padrão elementar é dado na Seção 7.

Para coletar os dados da saída (previsão do aluno) de um componente que faz parte de uma hipótese, o sistema dá para o aluno as entradas do componente. Para isso, os valores das variáveis de entrada do componente são calculados propagando-se os valores das entradas do programa correspondentes ao caso de teste escolhido para fazer o diagnóstico, até as portas de entrada do componente.

Interpretação dos dados

Com a previsão do aluno, o método de geração de hipóteses baseado em modelos é executado novamente, isto é, o sistema PROPAT_DEBUG é executado para as entradas e saídas do caso de teste mais as predições (observações) adicionais obtidas através do diálogo com o aluno, para gerar assim um conjunto igual ou menor de hipóteses.

6.5 Considerações finais

Nesse capítulo, foi apresentado o método de diagnóstico baseado em modelos aplicado a programas. Foram mostradas as conversões para se obter o modelo estrutural e comportamental baseado em valor a partir de um programa escrito na linguagem C restrita. Além disso, foi especificado o algoritmo de diagnóstico implementado no sistema PROPAT_DEBUG. No próximo capítulo será mostrado um exemplo completo de diagnóstico para um programa simples contendo uma falha que pode ter dois tipos de classificação: falha lógica estrutural ou falha lógica funcional.

CAPÍTULO 7

Exemplo de Diagnóstico de Programa

Nesse capítulo, mostramos um exemplo de diagnóstico para o programa **MAX_1**, dado na Tabela 7.2. Suponha que esse programa tenha sido construído por um aluno para resolver o seguinte problema (que chamaremos de MAX):

Dados dois números inteiros, imprimir o maior número multiplicado por 3.

No programa **MAX_1**, existem duas falhas possíveis no programa: a linha 8 que deveria ser $a > b$ ao invés de $a < b$ (falha simples do tipo funcional lógica F.3.1); ou as linhas 9 e 12 juntas que deveriam ser trocadas (falha múltipla do tipo estrutural lógica F.3.2). Vamos mostrar como o sistema de diagnóstico **ProPAT_deBUG** é capaz de detectar essas possíveis falhas.

Caso de Teste	Entradas	Saídas
1	a=5 b=4	greater=15
2	a=4 b=5	greater=15
3	a=5 b=5	greater=15

Tabela 7.1: Casos de teste usados para o problema MAX

7.1 Detecção de Sintomas

Inicialmente, o programa do aluno **MAX_1** é executado para os Casos de Teste da Tabela 7.1, apresentando um erro na variável de saída (*greater*) para os casos 1 e 2. Dessa forma, o sistema de diagnóstico é chamado para encontrar as falhas do programa que expliquem as observações de anormalidade, isto é, da saída errada $greater = 12$ para o caos Caso de Teste 2.

Programa: **MAX_1**	1.#include <stdio.h>
Linhas com erro: linha 8	2. int main () {
ou as linhas 9 e 12	3. int a,b,greater;
Descrição do erro:	4. printf("input a\n");
Na linha 8 a condição deveria ser $a > b$	5. scanf("%d",&a);
ou as linhas 9 e 12 estão trocadas.	6. printf("input b\n");
	7. scanf("%d",&b);
Caso de teste usado para	8. if($a < b$){
fazer o diagnóstico: 2	9. greater=3*a;
	10. }
	11. else{
Tipo de falha:	12. greater=3*b;
F.3.1 ou F.3.2	13. }
	14. printf("the greater is %d \n",greater);
	15. return 0;
	16.}

Tabela 7.2: Programa para o problema **MAX** com falha funcional na linha 8 (expressão lógica invertida) ou falha estrutural nas linhas 9 e 12 (linhas trocadas)

Selecionado o caso de teste que será usado para fazer o diagnóstico, o programa do aluno é então convertido num modelo de componentes e conexões (através do parser descrito na Seção 6.3.8).

7.2 Modelo Estrutural

Na Figura 7.1 é feita uma representação gráfica do modelo estrutural. Nessa figura, cada componente está associado a uma linha do programa. Por exemplo, os componentes C5 e C6 correspondem à linha 9 no programa do aluno, enquanto o componente C3 corresponde à linha 8, isto é, à instrução $if - then - else$ que inicia na linha 8 do programa. Note que os componentes C1 e C2, do tipo *Assignment*, foram usados para modelar os comandos *scanf* das linhas 5 e 7; e o comando C11, também do tipo *Assignment*, modela o comando *printf* da linha 14 do programa. O modelo não contém componentes para as linhas 4 e 6, uma vez que esses comando não envolve variáveis.

No modelo, existe uma conexão para cada variável do programa com nomes diferentes, cada vez que o valor da variável é alterado. Os nomes nas conexões para as varáveis modificadas são dados de acordo com o componente que a modificou. Por exemplo, a conexão b2 é o nome da variável b do programa após ser modificada pelo componente C2.

O modelo estrutural da Figura 7.1 é representado por um conjunto de axiomas A_s (chamados de *fatos* em

sistemas de produção). Esses axiomas (Tabela 7.3) definem as conexões entre componentes, isto é, as relações entre as portas de entrada e saída dos componentes. Os axiomas também associam essas conexões às variáveis ou constantes do programa. Por exemplo, o axioma A7.9 descreve que a porta de entrada $in_1(C3_then)$ do componente $C3_then$ está associada à conexão $a1'$ enquanto que o axioma A7.13 diz que a porta de entrada $in_2(C6)$ do componente $C6$ também está associada à mesma conexão. A partir desses dois axiomas pode-se inferir que os componentes $C3_then$ e $C6$ estão conectados através da conexão $a1'$.

Componente	Axioma	
C1	$in(C1) = input_a$	A7.1
	$out(C1) = a1$	A7.2
C2	$in(C2) = input_b$	A7.3
	$out(C2) = b2$	A7.4
C3	$condResult(C3) = aux4$	A7.5
	$in_1(C3) = a1$	A7.6
	$in_2(C3) = b2$	A7.7
	$out_1(C3) = greater3$	A7.8
C3_then	$in_1(C3_then) = a1'$	A7.9
	$in_2(C3_then) = b2'$	A7.10
	$out_1(C3_then) = greater5$	A7.11
C6	$in_1(C6) = const6$	A7.12
	$in_2(C6) = a1'$	A7.13
	$result(C6) = aux6$	A7.14
C5	$in(C5) = aux6$	A7.15
	$out(C5) = greater5$	A7.16
C3_else	$in_1(C3_else) = a1''$	A7.17
	$in_2(C3_else) = b2''$	A7.18
	$out_1(C3_else) = greater8$	A7.19
C9	$in_1(C9) = const9$	A7.20
	$in_2(C9) = b2''$	A7.21
	$result(C9) = aux9$	A7.22
C8	$in(C8) = aux9$	A7.23
	$out(C8) = greater8$	A7.24
C4	$in_1(C4) = a1$	A7.25
	$in_2(C4) = b2$	A7.26
	$result(C4) = aux4$	A7.27
C11	$in(C11) = greater3$	A7.28
	$out(C11) = output_greater$	A7.29

Tabela 7.3: Conjunto de axiomas A_s que define formalmente o modelo estrutural do programa do aluno.

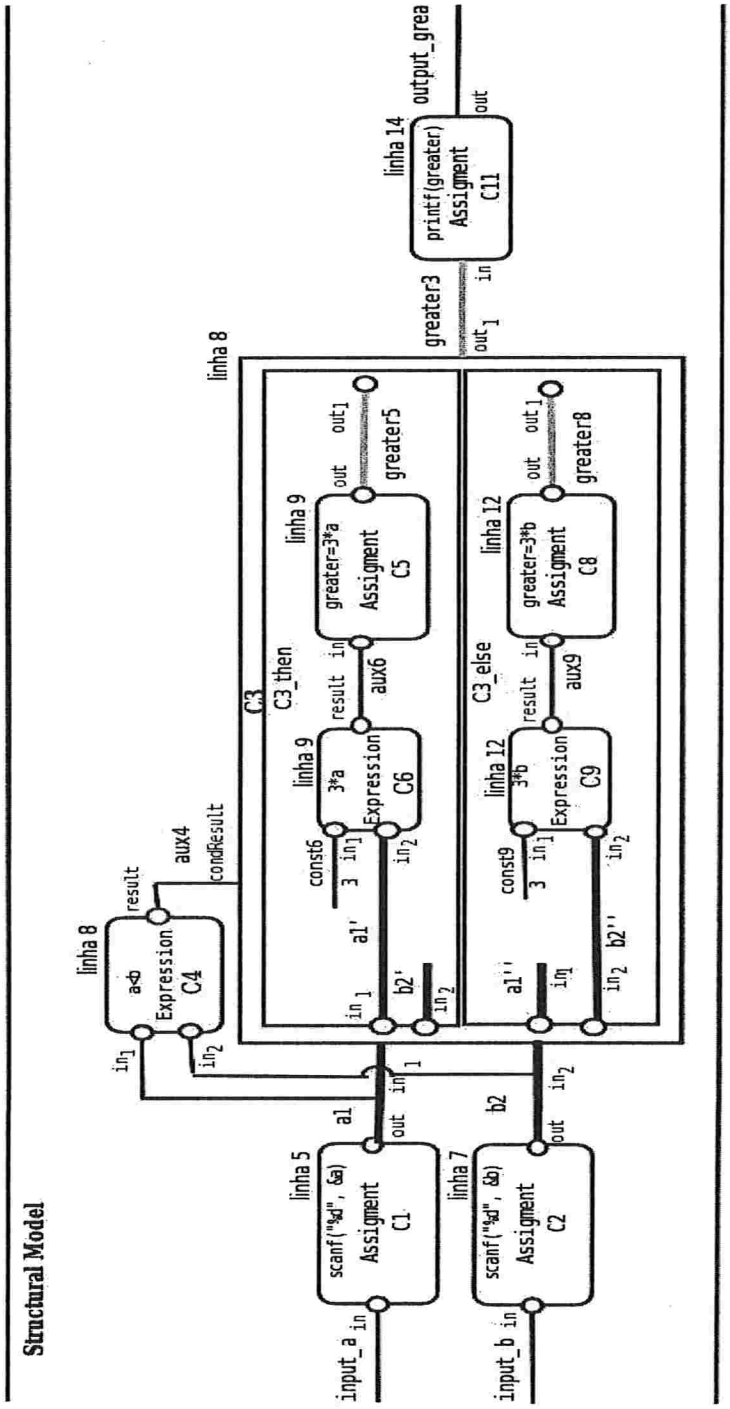


Figura 7.1: Modelo estrutural na forma gráfica do Programa *MAX_1*.

Em seguida, os valores do Caso de Teste 2 (entradas e saída), bem como as constantes do problema, são associadas às conexões de entrada e saída do modelo (input_a, input_b, output_greater, const6 e const9). Chamamos esse conjunto de axiomas de observação O, dado por:

$$input_a = 4 \tag{O7.1}$$

$$input_b = 5 \tag{O7.2}$$

$$output_greater = 15 \tag{O7.3}$$

$$const6 = 3 \tag{O7.4}$$

$$const9 = 3 \tag{O7.5}$$

7.3 Modelo Comportamental

A Tabela 7.4 descreve o conjunto de axiomas A_c que define uma parte do modelo comportamental do programa do aluno. Por exemplo, o axioma 7.35 descreve que o comportamento do componente $C3$ do tipo $if - then - else$, para o caso em que o ramo *then* é executado. Esse axioma diz que se o componente $C3$ estiver correto ($\neg AB(C3)$ e se o resultado da condição for *true* ($condResult(C3) = true$) implica que a saída do componente $C3$ deve ser igual à saída do componente $C3_then$ ($out_1(C3) = out_1(C3_then)$).

Nessa tabela não estão descritos todos os axiomas correspondentes às regras de inferência e de simulação (Seção 6.4.2) dos componentes, mas somente aquelas usadas no exemplo de geração de hipóteses dado a seguir. Por exemplo, nos axiomas que descrevem o comportamento do componente $C3$ estão somente as regras de simulação (implicações da entrada para a saída dos componentes), faltam as regras de inferência que propagam os valores no sentido inverso, ou seja, das saídas para as entradas.

7.4 Geração de Hipóteses

Para exemplificar o processo de geração de hipóteses será mostrado como é encontrado um conjunto de contribuintes, também chamado de conjunto conflito, fazendo-se uma chamada a um provador de teoremas. Dado um conjunto de observações, O; uma teoria que descreve o modelo estrutural e comportamental do programa do aluno, conjuntos A_s e A_c; e um conjunto de componentes considerados não falhos:

$COMP = \{\neg AB(C1), \neg AB(C2), \neg AB(C3), \neg AB(C3_then), \neg AB(C3_else), \neg AB(C4), \neg AB(C5), \neg AB(C6), \neg AB(C7), \neg AB(C8), \neg AB(C11)\}$,

deseja-se provar que esses conjuntos são logicamente consistentes, isto é:

Componente	Axioma	
C1	$\neg AB(C1) \rightarrow out(C1) = in(C1)$	A7.30
C2	$\neg AB(C2) \rightarrow out(C2) = in(C2)$	A7.31
C4	$\neg AB(C4) \rightarrow result(C4) = in_1(C4) \ < \ in_2(C4)$	A7.32
C3	$\neg AB(C3) \wedge condResult(C3) = true \rightarrow \ in_1(C3) = in_1(C3_then)$	A7.33
C3	$\neg AB(C3) \wedge condResult(C3) = true \rightarrow \ in_2(C3) = in_2(C3_then)$	A7.34
C3	$\neg AB(C3) \wedge condResult(C3) = true \rightarrow \ out_1(C3) = out_1(C3_then)$	A7.35
C6	$\neg AB(C6) \rightarrow result(C6) = in_1(C6) \ * \ in_2(C6)$	A7.36
C5	$\neg AB(C5) \rightarrow out(C5) = in(C5)$	A7.37
C11	$\neg AB(C11) \rightarrow in(C11) = out(C11)$	A7.38

Tabela 7.4: Axiomas do modelo comportamental.

$$O \cup A_s \cup \ A_c \cup \ COMP \nvdash \perp$$

Como nesse caso sabemos que existe uma inconsistência, vamos tentar encontrá-la, simulando o trabalho de um provador automático de teoremas. Da observação O7.1 e do axioma A7.1, temos que $in(C1) = 4$. Do axioma A7.30 e $\neg AB(C1)$, temos que $out(C1) = 4$ e de A7.2, inferimos que o valor da variável $a1$ é:

$$a1 = 4 \tag{S7.1}$$

Da observação O7.2 e do axioma A7.3, temos que $in(C2) = 5$. Do axioma A7.31 e $\neg AB(C2)$, temos que $out(C2) = 5$ e de A7.4, inferimos que o valor da variável $b1$ é:

$$b2 = 5 \tag{S7.2}$$

De S7.1 e S7.2, e a partir dos axiomas A7.25 e A7.26, temos que $in_1(C4) = 4$ e $in_2(C4) = 5$. Do axioma A7.32 e $\neg AB(C4)$, temos que $result(C4) = true$ e finalmente de A7.27 podemos inferir:

$$aux4 = true \tag{S7.3}$$

Do axioma A7.5 e da sentença S7.3, temos que:

$$condResult(C3) = true \tag{S7.4}$$

De A7.6 e S7.1, inferimos que $in_1(C3) = 4$ e de A7.33, S7.4 e $\neg AB(C3)$, temos que $in_1(C3_then) = 4$. Desse resultado considerando o axioma A7.9 temos que $a1' = 4$ e finalmente de A7.13 podemos inferir::

$$in_2(C6) = 4 \tag{S7.5}$$

Da observação O7.4 e do axioma A7.12 temos que:

$$in_1(C6) = 3 \tag{S7.6}$$

Das sentenças S7.5, S7.6, do axioma A7.36 e $\neg AB(C6)$ podemos inferir:

$$result(C6) = 12 \tag{S7.7}$$

Da sentença S7.7 e do axioma A7.14 temos que:

$$aux6 = 12 \tag{S7.8}$$

Da sentença S7.8 e do axioma A7.15, temos que $in(C5) = 12$. Do axioma A7.37 e $\neg AB(C5)$, temos que $out(C5) = 12$ e de A7.16 inferimos:

$$greater5 = 12 \tag{S7.9}$$

Da sentença S7.9 e do axioma A7.11, temos que $out_1(C3_then) = 12$ e de A7.35, S7.4 e $\neg AB(C3)$, temos que $out_1(C3) = 12$ e de A7.8 inferimos que o valor da variável greater3 é:

$$greater3 = 12 \tag{S7.10}$$

Da observação O7.3 e do axioma A7.29, temos $out(C11) = 15$ que com $\neg AB(C11)$ e A7.38 inferimos:

$$in(C11) = 15 \tag{S7.11}$$

Da sentença S7.11 e do axioma A7.28, inferimos que $greater3 = 15$, que gera uma inconsistência com S7.10 ($\perp$).

A discrepância foi encontrada na variável $greater3$ com valor 12 e 15. O conjunto de componentes usa-

dos para calcular $greater3 = 12$ foi {C1, C2, C3, C4, C5, C6} e o conjunto de componentes usados para calcular $greater3 = 15$ foi {C11}. Assim, o conjunto de componentes contribuintes para a inconsistência é $\{C1, C2, C3, C4, C5, C6, C11\}$, que correspondem às linhas do programa:

$$\{8, 9, 14, 7, 5\}$$

Esse conjunto será usado para rotular o primeiro nó do grafo criado pelo algoritmo de Reiter. A seguir, cada um desses contribuintes serão considerados falhos, isto é, para cada componente C do conjunto de contribuintes, considera-se $AB(C) = true$ no modelo comportamental e caso o provador de teoremas não encontre uma inconsistência, conclui-se que C é uma hipótese [1]. O algoritmo completo é executado e através de sucessivas chamadas ao provador de teoremas, como foi exemplificado acima, ele tenta encontrar conjuntos minimais de contribuintes, devolvendo para o programa **MAX_1** o seguinte diagnóstico:

$$HV = \{\{9\}, \{8\}, \{5\}, \{14\}, \{7, 12\}\}.$$

Como as linhas 5, 14 e 7 correspondem a comandos de leitura ou impressão, elas são excluídas da lista de hipóteses uma vez que não consideramos falhas para esses componentes[2]. A Tabela 7.5 mostra as hipóteses geradas pelo sistema de diagnóstico ordenadas pela estimativa da hipótese. Note que a hipótese $\{7, 12\}$ que pertence ao ramo *else* não executado do if tem o maior valor estimado de estarem corretas. Essa hipótese foi gerada pelo sistema de diagnóstico porque com a suspensão dos componentes C8 e C9 (linha 12) e o componente C2 (linha 7), não foi detectada uma inconsistência, o que implica que os componentes C8, C9 e C2 (linhas 7 e 12), formam uma hipótese.

Neste caso a heurística usada ajuda na detecção da falha, pois a explicação mais simples, linha 9, têm estimativa 0 que indica que essa é a linha menos provável de estar correta.

As Tabelas 7.6 e 7.7 mostram um exemplo de diálogo com o aluno para a discriminação de hipóteses. No inicio são mostradas ao aluno as hipóteses de menor estimativa de estarem corretas. O diálogo está baseado na documentação dos Padrões Elementares *ATRIBUIÇÃO* e *SELEÇÃO ALTERNATIVA* mostrados nas Figuras 7.2 7.3 respectivamente.

Demos apenas esse exemplo de script e em trabalhos futuros serão construídos scripts para o resto de padrões.

---

[1] Considerar o componente C como falho ($AB(C) = true$) é o mesmo que aplicar a técnica de suspensão. Uma vez que não existem regras de inferência ou de simulação em que $AB(C) = true$, os valores das variáveis não são propagados por esses componentes.

[2] No caso da hipótese $\{7, 12\}$, como as duas linhas devem estar falhas simultaneamente para que essa hipótese seja um diagnóstico, ao excluir o $scanf$ da linha 7, isso implica na exclusão da atribuição da linha 12.

Hipóteses (linhas) ordenadas segundo as estimativas	Estimativa da hipótese	Hipótese válida
9	0	Sim
8	0.5	Sim
5	0.5	Não
14	0.5	Não
12, 7	1.1666	Não

Tabela 7.5: Hipóteses geradas pelo sistema de diagnóstico para o Programa *MAX_1*.

7.5 Considerações finais

Nesse capítulo foi apresentado um exemplo de diagnóstico para um programa simples. A partir do programa, foi mostrado como construir um conjunto de axiomas A_s e A_c, que definem formalmente o modelo estrutural e comportamental do programa do aluno. Dado que nesse exemplo existe falha que provoca uma inconsistência no modelo formal, foi simulado o trabalho de um provador automático de teoremas que permite encontrar um dos conjuntos minimais de contribuintes. Além disso, foi mostrado passo a passo, a geração do conjunto de hipóteses pelo algoritmo de Reiter, apartir dos conjuntos de contribuintes devolvidos pelas sucessivas chamadas ao provador de teoremas. Finalmente é mostrado o diálogo com o aluno para discriminar as hipóteses. O script usado durante o diálogo com o aluno foi mostrado para dar um exemplo de como construir scripts de diálogos a partir de Padrões Elementares.

No próximo capítulo o sistema PROPAT_DEBUG será avaliado para um conjunto de programas com falhas lógicas estruturais e funcionais.

nome do padrão *Atribuição*

estrutura da solução 1: sintaxe

< nome-da-variável> = <expressão>

estrutura da solução 2: semântica

A expressão é avaliada, e uma cópia do valor resultante é armazenada na posição de memória referenciada pelo nome da variável.

aplicabilidade / objetivo

Você quer guardar um valor para uso posterior, dando-lhe um nome.

Figura 7.2: Parte do documento do Padrão Elementar Atribuição.

Sistema Tutor:	As linhas mais prováveis de estarem com erro são: 1. ***linha 9*** 2. ***linha 8*** Para descobrir qual delas é a hipótese correta de falha em seu programa vamos investigar primeiro a ***linha*** 9. A ***linha 9*** corresponde ao Padrão Elementar *ATRIBUIÇÃO*: ***você quer armazenar na variável greater o resultado de uma expressão***: $3 * a$ PERGUNTA: ***Para a=4, qual é o valor da variável greater após executar a linha 9?***
Aluno: ou **Aluno:**	15 ======> a falha foi detectada 12
Sistema Tutor:	De fato, o seu programa calcula o mesmo valor. No entanto, como ainda existe uma inconsistência, para resolvê-la devemos investigar a hipótese da linha 9 em mais detalhes. A linha 9 faz parte do Padrão Elementar *SELEÇÃO ALTERNATIVA*, sendo essa linha pertencente ao conjunto de ações da primeira parte, isto é, da parte que é executada quando a condição for verdadeira. Para $a = 4$ e $b = 5$, a condição $a < b$ é verdadeira e portanto a linha 9 é de fato executada. É essa a ação que você quer executar quando $a < b$ for verdadeira?
Aluno: ou **Aluno:**	NÂO ======> a falha foi detectada SIM
Sistema Tutor:	Nesse caso, como ainda existe uma inconsistência, para resolvê-la devemos investigar a próxima hipótese de falha na linha 8. ======> O aluno não conseguiu perceber a falha com a primeira hipótese. Talvez ele consiga perceber a falha com a segunda hipótese.

Tabela 7.6: Diálogo para a discriminação das hipóteses para o Programa *MAX_1*.

	Essa linha, corresponde à *condição de uma Seleção Alternativa.* Note que essa condição determina qual das duas partes da Seleção Alternativa será executada. Sendo a condição uma hipótese de falha, pode ser que este sendo executado o conjunto errado de ações. PERGUNTA: ***qual é o valor que você esperava para a condição*** $a < b$ ***para*** $a = 4$ ***e*** $b = 5$***?***
Aluno:	*verdadeiro*
Sistema Tutor:	De fato, o seu programa calcula o mesmo valor. No entanto, como ainda existe uma inconsistência, para resolvê-la devemos investigar a hipótese da linha 8 em mais detalhes PERGUNTA: ***qual é o valor que você esperava para a greater depois da linha 13 para*** $a = 4$ ***e*** $b = 5$***?***
Aluno:	15 ======> a falha foi detectada

Tabela 7.7: Diálogo para a discriminação das hipóteses para o Programa *MAX_1* (continuação).

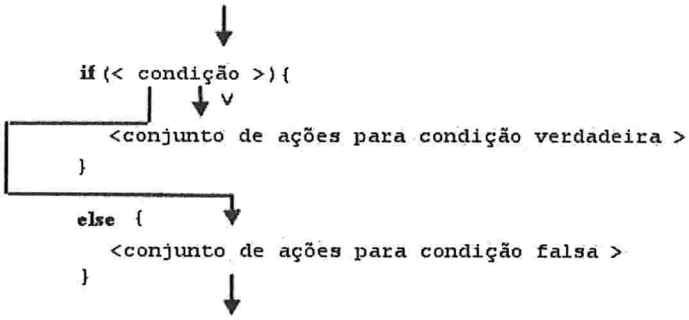
nome do padrão *Seleção Alternativa*

estrutura da solução 1: sintaxe

```
if (< condição >) {
   <conjunto de ações para condição verdadeira >
}
else {
   <conjunto de ações para condição falsa >
}
```

estrutura da solução 2: semântica

A condição é testada. Se ela for satisfeita, então o conjunto de ações da primeira parte será executado. Caso contrário, o conjunto de ações da segunda parte é que será executado.



aplicabilidade / objetivo

Você quer selecionar para executar um entre dois conjuntos de ações alternativos, baseado em uma condição.

Figura 7.3: Parte do documento do Padrão Elementar Seleção Alternativa.

CAPÍTULO 8

Resultados experimentais

Neste capítulo, é feita uma avaliação de desempenho do sistema PROPAT_DEBUG para 28 programas com diferentes tipos de falhas. Essa avaliação será em termos do resultado do diagnóstico, isto é, da precisão e cobertura do conjunto de hipóteses devolvido pelo sistema. Apesar de não ser feita aqui uma avaliação em termos de aprendizagem envolvendo alunos num curso introdutório, a avaliação das hipóteses devolvidas traz resultados que indicam um aspecto essencial num sistema de diagnóstico de programas: apresentar um número pequeno de hipóteses sobre o erro do aluno.

8.1 Metodologia

Para avaliar o sistema de diagnóstico proposto, PROPAT_DEBUG, foram selecionados 7 problemas com nível de dificuldade crescente. Para cada problema foram construídos 4 programas introduzindo falhas típicas segundo a classificação dada na Seção 2.3. Uma vez que todos os programas foram compilados com sucesso e todos apresentam falhas que causam consequência na saída, essa classificação se refere basicamente a falhas lógicas e semânticas.

Na Tabela 8.1 são definidos os problemas selecionados para essa avaliação e na Tabela 8.2 são dados os casos de teste para cada problema. Os problemas envolvem ações simples, seleção simples, seleção alternativa, seleção seqüencial, repetição contada e repetição com sentinela

Em *http : //www.ime.usp.br/ ~ kvd/mestrado/diag_exemplos.pdf*, são descritos todos os programas (vinte e oito) detalhando as falhas com suas respectivas classificações e as hipóteses geradas pelo sistema de diagnóstico. A Tabela 8.3 apresenta um resumo dos resultados obtidos ao aplicar o sistema de diagnóstico nos programas dos alunos, onde:

- **L** é o número total de linhas do programa do aluno;

- **FP** indica os números das linhas com falha. No caso de falhas estruturais, são descritas as sentenças que faltam no programa;
- **C** é o número de componentes criados no modelo de componentes e conexões;
- **H** é o número de hipóteses válidas geradas pelo sistema de diagnóstico;
- **HV** é a lista das hipóteses válidas ordenadas pela estimativa da hipótese;
- **P** é a posição da falha na lista ordenada de hipóteses;
- **T** é o tempo em segundos que demorou o sistema de diagnóstico para gerar as hipóteses (inclui o tempo para criar o modelo, o cálculo da heurística e o tempo do cálculo das hipóteses propriamente ditas);
- **D** indica se o sistema de diagnóstico descobriu ou não as falhas;
- **FTP** indica o nome da falha e
- **CF** são dados os tipos de falha que foram introduzidas nos programas segundo a classificação de falhas lógicas e semânticas da Seção 2.3.

8.1.1 Medidas usadas na Avaliação do Diagnóstico

A avaliação do sistema de diagnóstico objetiva, basicamente, medir a capacidade do sistema em devolver um conjunto de hipóteses que inclua as falhas reais do programa e que não devolva um número muito grande de hipóteses incorretas (isto é, hipóteses que podem explicar as observações mas que não são falhas reais). Para isso, foram definidas as medidas de *precisão* e *cobertura* que são geralmente usadas na área de recuperação de informação [VR79]. Os conjuntos usados nessas medidas foram redefinidos para a avaliação do desempenho de sistemas de diagnóstico como sendo:

FP={conjunto de falhas no programa do aluno}

HV={conjunto das k primeiras hipóteses devolvidas pelo sistema de diagnóstico ordenadas segundo a estimativa da hipótese (Seção 6.4.3) }

Precisão (PRE) é a razão entre o número de hipóteses corretas devolvidas pelo sistema ($|FP \cap HV|$) e o número total de hipóteses devolvidas pelo sistema.

$$PRE = \frac{|FP \cap HV|}{|HV|} \tag{8.1}$$

ou seja, $0 \leq PRE \leq 1$. Essa razão expressa a precisão do sistema de diagnóstico em encontrar a solução. $PRE = 1$ significa que todas as hipóteses geradas pelo sistema de diagnóstico são falhas reais, isto é, se

$h \in HV \rightarrow h \in FP$. Isso pode ser inferido de:

$$\frac{|FP \cap HV|}{|HV|} = 1 \Rightarrow FP \cap HV = HV \Rightarrow HV \subseteq FP \Rightarrow h \in HV \rightarrow h \in FP \tag{8.2}$$

$PRE = 0$ significa que $|FP \cap HV| = 0$, ou seja, o sistema de diagnóstico não foi capaz de encontrar nenhuma das falhas do programa. Valores de PRE próximos de 1 indica que o diagnóstico é mais precioso. Valores PRE mais próximo de 0 indica que o diagnóstico encontrado é menos preciso pois devolve um grande número de hipóteses que não contêm a falha real.

Cobertura (COB) é a razão entre o número de hipóteses corretas devolvidas pelo sistema e o número de falhas no programa do aluno.

$$COB = \frac{|FP \cap HV|}{|FP|} \tag{8.3}$$

ou seja, $0 \leq COB \leq 1$. Essa razão expressa a cobertura do sistema de diagnóstico em encontrar a solução. $COB = 1$ significa que todas as falhas reais foram descobertas pelo sistema de diagnóstico, isto é, se $h \in FP \rightarrow h \in HV$. Isso pode ser inferido da seguinte forma:

$$\frac{|FP \cap HV|}{|FP|} = 1 \Rightarrow FP \cap HV = FP \Rightarrow FP \subseteq HV \Rightarrow h \in FP \rightarrow h \in HV \tag{8.4}$$

$COB = 0$ significa que $|FP \cap HV| = 0$, ou seja, o sistema de diagnóstico não foi capaz de encontrar nenhuma das falhas do programa. Quanto mais próximo de 1, COB indica uma melhor cobertura do diagnóstico com relação às falhas do programa. Quando o valor de COB é mais próximo de 0, indica uma menor cobertura do diagnóstico sobre as falhas reais do programa.

O que é esperado de um sistema de diagnóstico em termos de desempenho é um compromisso entre precisão e cobertura, por exemplo, no programa de exemplo do Capítulo 7, se o diagnóstico do programa do aluno for **todas as linhas** do programa, tem-se $COB = 1$, porém, a precisão do diagnóstico seria $PRE = \frac{1}{16}$.

Medida-F é uma medida que combina precisão e cobertura, definida como:

$$Medida - F = \frac{1}{\frac{\alpha}{PRE} + \frac{1-\alpha}{COB}} \tag{8.5}$$

Sendo α, $0 < \alpha < 1$, usado para indicar a importância relativa da precisão para o usuário. Uma melhora na Medida-F indica uma melhora tanto da precisão quanto na cobertura. Para $\alpha = 0.5$ dá-se a mesma importância

para precisão e a cobertura, resultando na média harmônica, isto é:

$$Medida - F = \frac{2 * PRE * COB}{PRE + COB} \tag{8.6}$$

Nome do Problema	Enunciado
MARATONA	Em uma maratona, o tempo é medido em horas, minutos, segundos e a distância percorrida em metros. Calcular a velocidade dos participantes em metros/segundos (todas as variáveis são inteiras e maiores que zero). O programa também deverá calcular o peso do atleta em gramas a partir de seu peso em kilogramas.
MAX	Dados dois números inteiros, imprimir o maior deles multiplicado por 3.
MEDIANA	Dados três números inteiros, imprimir a mediana, isto é, o número que está entre o menor e o maior.
TRIÂNGULO	Dados três números inteiros lidos em ordem crescente. Determinar se os números correspondem aos lados de um triângulo (a soma dos menores lados tem que ser maior que o lado maior). Se não correspondem imprimir 0. Caso contrário, imprimir 1 se é um triângulo escaleno, 2 se é isósceles e 3 se é eqüilátero.
SOMA_NATURAIS	Dado um número inteiro positivo n, calcular a soma dos n primeiros números naturais.
SOMA_MULTIPLOS	Dado um número inteiro positivo n, calcular a soma dos múltiplos de 5, de 100 até n.
SOMA_N_NUMEROS	Dado um número inteiro positivo n, e uma seqüência de n números inteiros, determinar a soma dos elementos da seqüência.

Tabela 8.1: Enunciado dos problemas usados pelo sistema de diagnóstico.

Nome do Problema	Caso de teste	Entradas	Saídas
MARATONA	1	h=1 m=5 s=100 me=8000 kg=55	v=2 g=55000
	2	h=1 m=0 s=100 me=3700 kg=60	v=1 g=60000
	3	h=2 m=10 s=0 me=23400 kg=70	v=3 g=70000
MAX	1	a=5 b=4	greater=15
	2	a=4 b=5	greater=15
	3	a=5 b=5	greater=15
MEDIANA	1	x=1 y=1 z=5	m=1
	2	x=1 y=2 z=3	m=2
	3	x=3 y=2 z=1	m=2
	4	x=5 y=5 z=5	m=5
	5	x=5 y=3 z=4	m=4
	6	x=2 y=1 z=3	m=2
TRIÂNGULO	1	a=2 b=2 c=3	t=2
	2	a=1 b=1 c=3	t=0
	3	a=2 b=5 c=5	t=2
	4	a=3 b=6 c=8	t=1
	5	a=2 b=3 c=8	t=0
	6	a=3 b=3 c=3	t=3
SOMA_NATURAIS	1	n=2	sum=3
	2	n=1	sum=1
	3	n=0	sum=0
	4	n=4	sum=10
SOMA_MULTIPLOS	1	n=0	sum=0
	2	n=100	sum=100
	3	n=105	sum=205
	4	n=112	sum=315
SOMA_N_NUMEROS	1	n=4 number 6 2 5 4	sum=17
	2	n=1 number 1	sum=1
	3	n=2 number 1 3	sum=4
	4	n=3 number 5 0 3	sum=8

Tabela 8.2: Casos de teste usados para testar o sistema de diagnóstico

Programa	L	FP	C	H	HV	P	T	D	FTP	CF
1 MARATONA_1	20	{14}	16	2	{14}{15}	1	1	S		F.3.1
2 MARATONA_2	20	{17}	16	1	{17}	1	1	S		F.3.1
3 MARATONA_3	20	{15,17}	16	1	{15,17}	1	1	S		F.3.1
4 MARATONA_4	22	{16}	17	2	{16},{17}	1	1	S		F.3.1 F.2.1
5 MAX_1	16	{8} ({9,12})	9	2	{9}{8}	2	1	S		F.3.1 F.3.2 F.2.1
6 MAX_2	16	{13} ({9,10})	9	2	{13}{9}	1	1	S		F.3.1 F.2.1
7 MAX_3	15	falta if que inclua a linha 12	12	1	{12}	1	1	S	Condicional4	F.3.1 F.3.2 F.2.1
8 MAX_4	14	{10*}=falta else do if da linha 10	10	3	{10}{5}{10*}	2	1	S		F.3.2 F.2.1
9 MEDIANA_1	27	{16}	21	2	{15(if que inclui a linha 16)}{11}	1	4	S		F.3.1
10 MEDIANA_2	20	{11*}=falta else do if da linha 11	15	3	{11*}{11}{10}	1	3	S		F.3.2 F.2.2
11 MEDIANA_3	13	na linha 10 deveria usar seleção seqüêncial	8	1	{10}	1	2	S	Interpretação1	F.4
12 MEDIANA_4	25	faltam comparações	26	2	{15}{15*}	2	4	S		F.3.2 F.2.2
13 TRIÂNGULO_1	30	{20}	19	5	{20}{19}{12}{11}{26, 10}(19,12,11 incluem a linha 20)	1,2,3,4	4	S		F.3.1
14 TRIÂNGULO_2	26	{11} (falta um if)	30	3	{11}{12}{12*}	1	3	S		F.3.2 F.2.2 F.3.1
15 TRIÂNGULO_3	29	{15, 18, 21, 24, 15* ,18* ,21* ,24*}	35	6	{19}{24*}{18}{21}{24} {21*}	2,3,4	4	S	Paralelismo	F.3.2 F.2.2
16 TRIÂNGULO_4	29	{12}	31	3	{12}{13}{10}	1	4	S		F.3.1 F.2.2
17 SOMA_NATURAIS_1	14	{9}	20	3	{8}{9}{6}	1	4	S	Variável1	F.3.1 F.2.1
18 SOMA_NATURAIS_2	14	{8}	12	4	{8}{9}{6}{7}	1	4	S	ExpLaço4	F.3.1 F.2.1
19 SOMA_NATURAIS_3	14	{11}	16	5	{8}{9}{11}{6}{7}	2	3	S	Laço1	F.3.2 F.2.1
20 SOMA_NATURAIS_4	14	{8}	13	4	{9}{8}{6}{7}	2	3	S	LaçoCondicional1	F.3.2 F.2.1
21 SOMA_MULTIPLOS_1	18	{9}	19	2	{9}{9*}	1	6	S	ExpLaço4	F.3.1 F.2.1
22 SOMA_MULTIPLOS_2	14	{9,10,7}	20	5	{8}{9}{10}{6}{7}	1,2	4	S	ExpLaço1	F.3.2 F.2.2
23 SOMA_MULTIPLOS_3	14	falta laço que inclua as linhas 9 e 10	8	3	{9}{6}{7}	1	2	S	Laço3 Laço4	F.3.2 F.2.1
24 SOMA_MULTIPLOS_4	14	{7,10}	13	6	{9}{11}{10} {6} {7} {8,9*}	1,2	7	S	Variável1	F.3.1 F.2.1 C.2
25 SOMA_N_NUMEROS_1	16	{11}	22	6	{8} {11} {6} $\{8^{*1},7\}\{8^{*2},7\}\{8^{*2},12\}$	1	5	S	Variável1	F.3.1 F.2.1
26 SOMA_N_NUMEROS_2	18	{8,9,10,11,12}=deveria usar repetição contada	14	3	{10} {11} {6} (10 inclui linha 11 e 12)	1	2	S		F.3.2 F.2.2
27 SOMA_N_NUMEROS_3	18	{8,9,10,11,12}=deveria usar repetição contada	14	3	{10} {11} {6} (10 inclui linha 11 e 12)	1	2	S		F.3.2 F.2.2
28 SOMA_N_NUMEROS_4	16	{11}	20	7	{8} {11,6} $\{6,7\}\{8^{*1},11\}\{8^{*1},7\}$ $\{8^{*2},6,12\}$ $\{8^{*1},8^{*2},12\}$	2,3	3	S	Variável1	F.3.1 F.2.1

Tabela 8.3: Resultados de aplicar o diagnóstico para diferentes programas de alunos. "*"indica a falta do else no condicional ou a falta de uma execução a mais do laço.

Prog	$\lvert FP \cap HV_1 \rvert$	$\lvert FP \cap HV_2 \rvert$	$\lvert FP \cap HV_3 \rvert$	$\lvert HV_1 \rvert$	$\lvert HV_2 \rvert$	$\lvert HV_3 \rvert$	$\lvert FP \rvert$	PRE_1	PRE_2	PRE_3	COB_1	COB_2	COB_3
1	1.00	1.00	1.00	2.00	2.00	2.00	1.00	0.50	0.50	0.50	1.00	1.00	1.00
2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4	1.00	1.00	1.00	2.00	2.00	2.00	1.00	0.50	0.50	0.50	1.00	1.00	1.00
5	0.00	1.00	1.00	1.00	2.00	2.00	1.00	0.00	0.50	0.50	0.00	1.00	1.00
6	1.00	1.00	1.00	1.00	2.00	2.00	1.00	1.00	0.50	0.50	1.00	1.00	1.00
7	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
8	1.00	1.00	1.00	2.00	2.00	3.00	1.00	0.50	0.50	0.33	1.00	1.00	1.00
9	1.00	1.00	1.00	1.00	2.00	2.00	1.00	1.00	0.50	0.50	1.00	1.00	1.00
10	1.00	1.00	1.00	1.00	3.00	3.00	1.00	1.00	0.33	0.33	1.00	1.00	1.00
11	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
12	1.00	1.00	1.00	1.00	2.00	2.00	1.00	1.00	0.50	0.50	1.00	1.00	1.00
13	1.00	1.00	1.00	1.00	2.00	3.00	1.00	1.00	0.50	0.33	1.00	1.00	1.00
14	1.00	1.00	1.00	2.00	2.00	3.00	1.00	0.50	0.50	0.33	1.00	1.00	1.00
15	0.00	0.13	0.63	1.00	2.00	5.00	1.00	0.00	0.06	0.13	0.00	0.13	0.63
16	1.00	1.00	1.00	3.00	3.00	3.00	1.00	0.33	0.33	0.33	1.00	1.00	1.00
17	1.00	1.00	1.00	2.00	2.00	3.00	1.00	0.50	0.50	0.33	1.00	1.00	1.00
18	1.00	1.00	1.00	1.00	2.00	4.00	1.00	1.00	0.50	0.25	1.00	1.00	1.00
19	0.00	0.00	1.00	2.00	2.00	5.00	1.00	0.00	0.00	0.20	0.00	0.00	1.00
20	0.00	1.00	1.00	1.00	4.00	4.00	1.00	0.00	0.25	0.25	0.00	1.00	1.00
21	1.00	1.00	1.00	1.00	2.00	2.00	1.00	1.00	0.50	0.50	1.00	1.00	1.00
22	0.67	1.00	1.00	3.00	3.00	5.00	1.00	0.22	0.33	0.20	0.67	1.00	1.00
23	0.50	0.50	0.50	3.00	3.00	3.00	1.00	0.17	0.17	0.17	0.50	0.50	0.50
24	0.50	0.50	0.50	3.00	3.00	3.00	1.00	0.17	0.17	0.17	0.50	0.50	0.50
25	0.00	1.00	1.00	3.00	3.00	3.00	1.00	0.00	0.33	0.33	0.00	1.00	1.00
26	0.40	0.40	0.40	3.00	3.00	3.00	1.00	0.13	0.13	0.13	0.40	0.40	0.40
27	0.40	0.40	0.40	3.00	3.00	3.00	1.00	0.13	0.13	0.13	0.40	0.40	0.40
28	0.00	0.50	0.50	1.00	3.00	3.00	1.00	0.00	0.17	0.17	0.00	0.50	0.50

Tabela 8.4: Medidas de precisão e cobertura para os diagnósticos dos 28 programas.

Prog.	$MF_1(\alpha = 0.5)$	$MF_2(\alpha = 0.5)$	$MF_3(\alpha = 0.5)$	$MF_3(\alpha = 0.3)$
1	0.67	0.67	0.67	0.77
2	1.00	1.00	1.00	1.00
3	1.00	1.00	1.00	1.00
4	0.67	0.67	0.67	0.77
5	0.00	0.67	0.67	0.77
6	1.00	0.67	0.67	0.77
7	1.00	1.00	1.00	1.00
8	0.67	0.67	0.50	0.63
9	1.00	0.67	0.67	0.77
10	1.00	0.50	0.50	0.63
11	1.00	1.00	1.00	1.00
12	1.00	0.67	0.67	0.77
13	1.00	0.67	0.50	0.63
14	0.67	0.67	0.50	0.63
15	0.00	0.08	0.21	0.28
16	0.50	0.50	0.50	0.63
17	0.67	0.67	0.50	0.63
18	1.00	0.67	0.40	0.53
19	0.00	0.00	0.33	0.45
20	0.00	0.40	0.40	0.53
21	1.00	0.67	0.67	0.77
22	0.33	0.50	0.33	0.45
23	0.25	0.25	0.25	0.31
24	0.25	0.25	0.25	0.31
25	0.00	0.50	0.50	0.63
26	0.20	0.20	0.20	0.25
27	0.20	0.20	0.20	0.25
28	0.00	0.25	0.25	0.31

Tabela 8.5: Medida-F para os diagnósticos dos 28 programas.

8.2 Análise dos Resultados

As medidas de precisão, cobertura e medida-F foram calculadas para o conjunto HV composto pelas k primeiras hipóteses devolvidas pelo sistema de diagnóstico ordenadas de acordo com a estimativa da hipótese, com $k \in \{1, 2, 3\}$.

A Tabela 8.4 mostra as medidas calculadas para os 28 programas, sendo que as colunas HV_k, PRE_k e COB_k indicam, respectivamente, o conjunto HV e as medidas PRE e COB para os diferentes valores de k. A Tabela 8.5 mostra as medidas-F para $\alpha = 0.5$, com $k \in \{1, 2, 3\}$, e para $\alpha = 0.3$, com $k = 3$. Conforme a definição, quanto mais próximo de 1 a medida-F for, melhor é o compromisso entre as medidas de precisão e a cobertura.

As Figuras 8.1, 8.2 e 8.3 mostram as medidas-F para o conjunto de hipóteses devolvidas para k=1, k=2 e k=3, respectivamente, com $\alpha = 0.5$ (isto é, atribuindo-se a mesma importância para as medidas de precisão e cobertura). Como pode ser observado, a Figura 8.1 mostra que para k=1, a medida-F apresenta valores muito oscilantes. Nessa figura, os casos com valor 0, significa que a falha não estava na primeira posição da lista

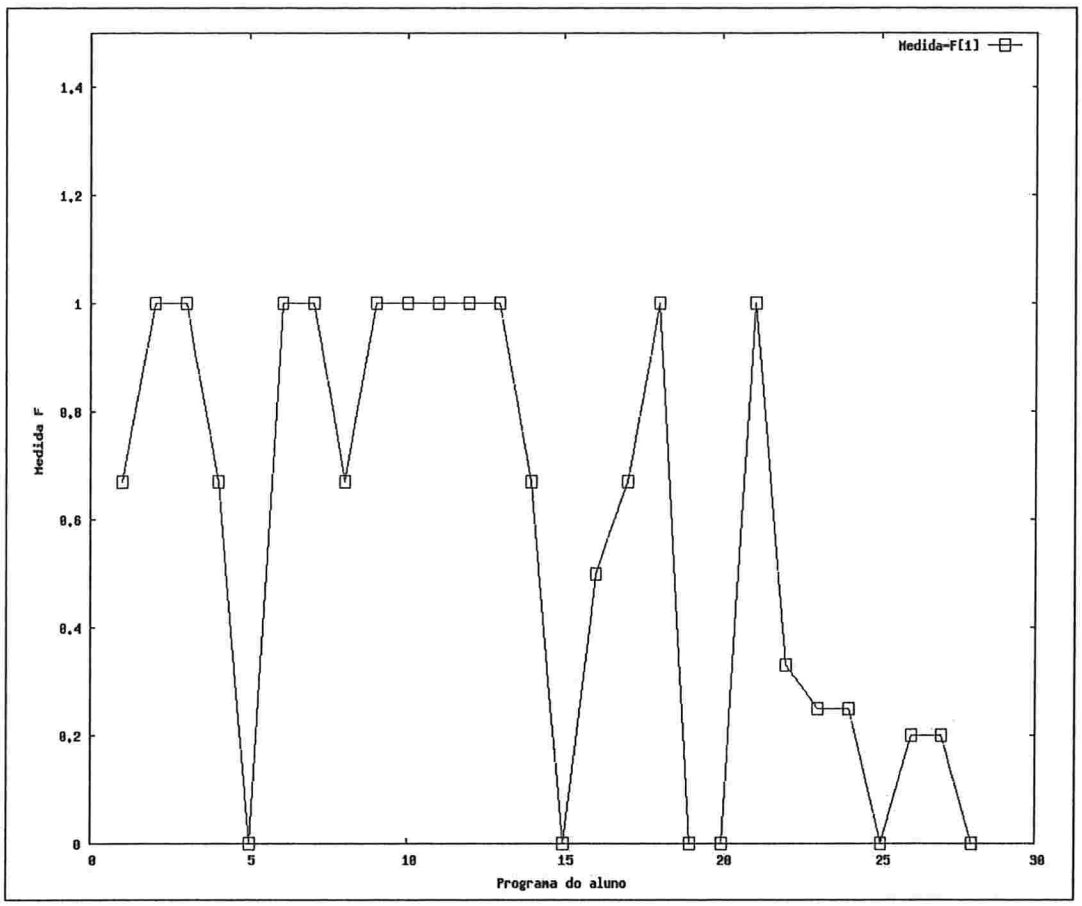


Figura 8.1: Medida F para k=1 e $\alpha = 0.5$.

ordenada de hipóteses, o que é refletido nas medidas de cobertura e precisão.

Considerando-se $k = 2$ (Figura 8.2), obtém-se valores para medida-F mais próximos de 1 com um número menor de valores iguais a zero. Isso pode ser explicado pelo fato da maioria das falhas pertencerem à primeira ou segunda posição da lista de hipóteses (coluna P da Tabela 8.3). Considerando $k = 3$, observamos que os valores da medida-F (Figura 8.3) apresentam uma melhora no compromisso entre precisão e cobertura. Como esses resultados ainda não estão muito próximos de 1, foi feita uma análise da importância relativa entre a precisão e cobertura do diagnóstico, variando-se o valor de α. A Figura 8.4 mostra os valores da medida-F para $k = 3$ e $\alpha = 0.3$, isto é, diminuindo-se a importância da precisão com relação à medida de cobertura. Isso resultou numa pequena melhora nos valores da medida-F.

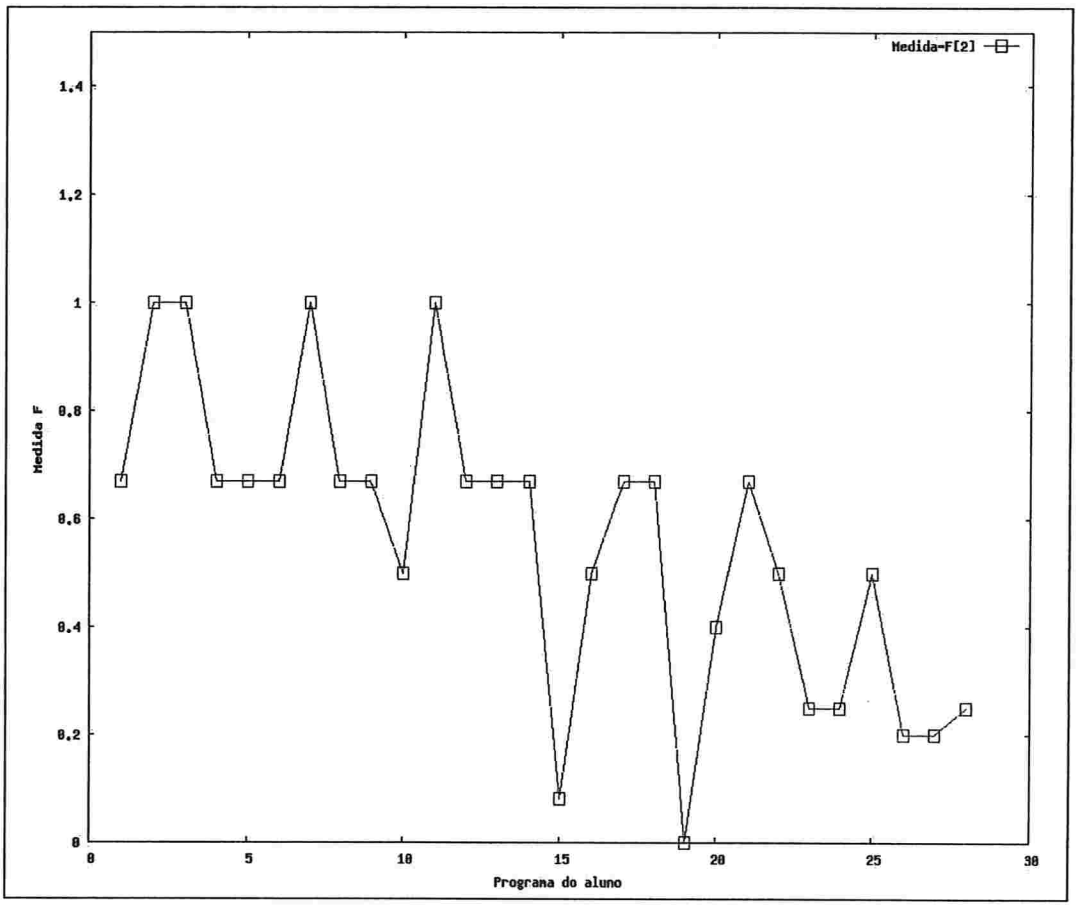


Figura 8.2: Medida F para k=2 e $\alpha = 0.5$.

A maioria dos problemas (64%) tiveram valores para medida-F entre 0.63 e 1. Fazendo-se uma análise dos programas com valores de medida-F inferiores a 0.63, 7 deles apresentaram falhas estruturais que são as mais difíceis de serem detectadas. Excluindo-se esses programas tem-se 86% dos programas restantes com valores de medida-F acima de 0.63. Os pontos pretos na Figura 8.4 indicam os programas com falhas estruturais.

É importante notar que na grande maioria dos programas testados, a medida de cobertura foi alta para um número relativamente baixo de hipóteses geradas. Isso pode ser visto na Figura 8.5. Na próxima seção explicamos porque uma boa cobertura pode ser mais importante do que a melhora na medida de precisão.

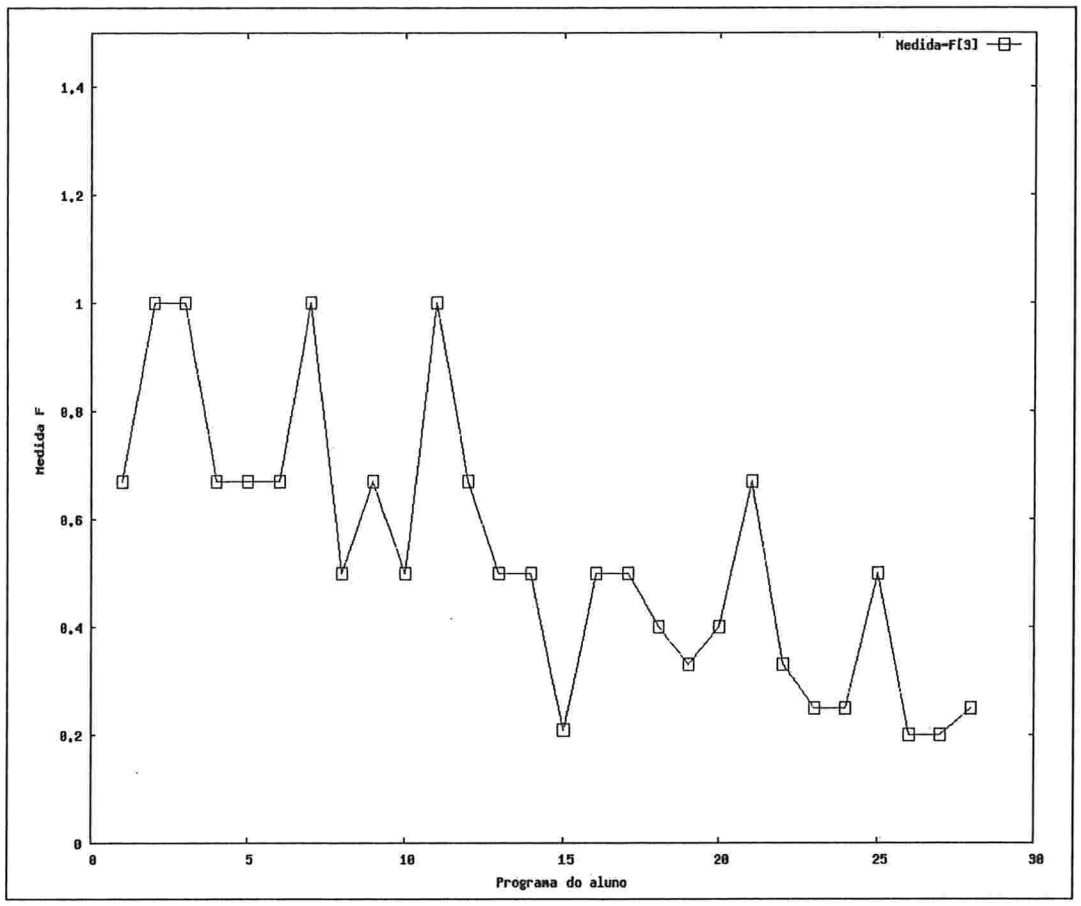


Figura 8.3: Medida F para k=3 e $\alpha = 0.5$.

8.2.1 Falhas múltiplas alternativas

Um aspecto interessante no diagnóstico de programas baseado em modelos é que todas as hipóteses geradas explicam logicamente as observações de falha, o que pode ser comprovado para os 28 programas testados. Isso quer dizer que dada uma hipótese é possível corrigir o programa para que as saídas sejam corretas, para um dado caso de teste. Em alguns casos, ainda que essa não seja a falha original do aluno, é possível corrigir o programa de modo que esse funcione para todos os casos de teste. Por exemplo, para o programa MAX_1 cujo diagnóstico para o caso de teste 2 (Tabela 7.1) foi detalhado no Capítulo 7, existem 2 falhas possíveis: {9 e 12} ou {8}. Sendo que o sistema de diagnóstico devolveu as seguintes hipóteses de falha:

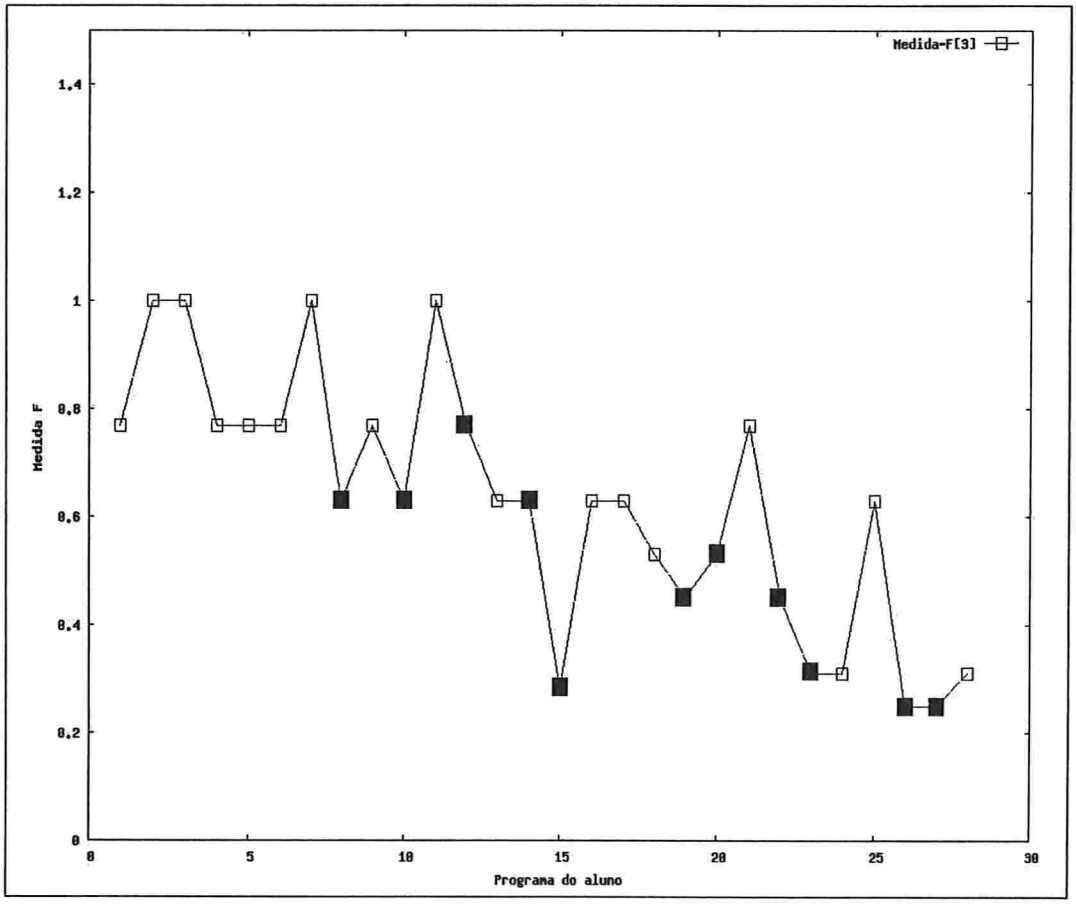


Figura 8.4: Medida F para k=3 e $\alpha = 0.3$.Os pontos pretos indicam programas com falhas lógicas estruturais.

Hipótese 1: linha 8
Hipótese 2: linha 9

Para um outro caso de teste (caso 1 da Tabela 7.1) as hipóteses de falha devolvidas pelo sistema de diagnóstico foram:

Hipótese 1: linha 8
Hipótese 2: linha 12

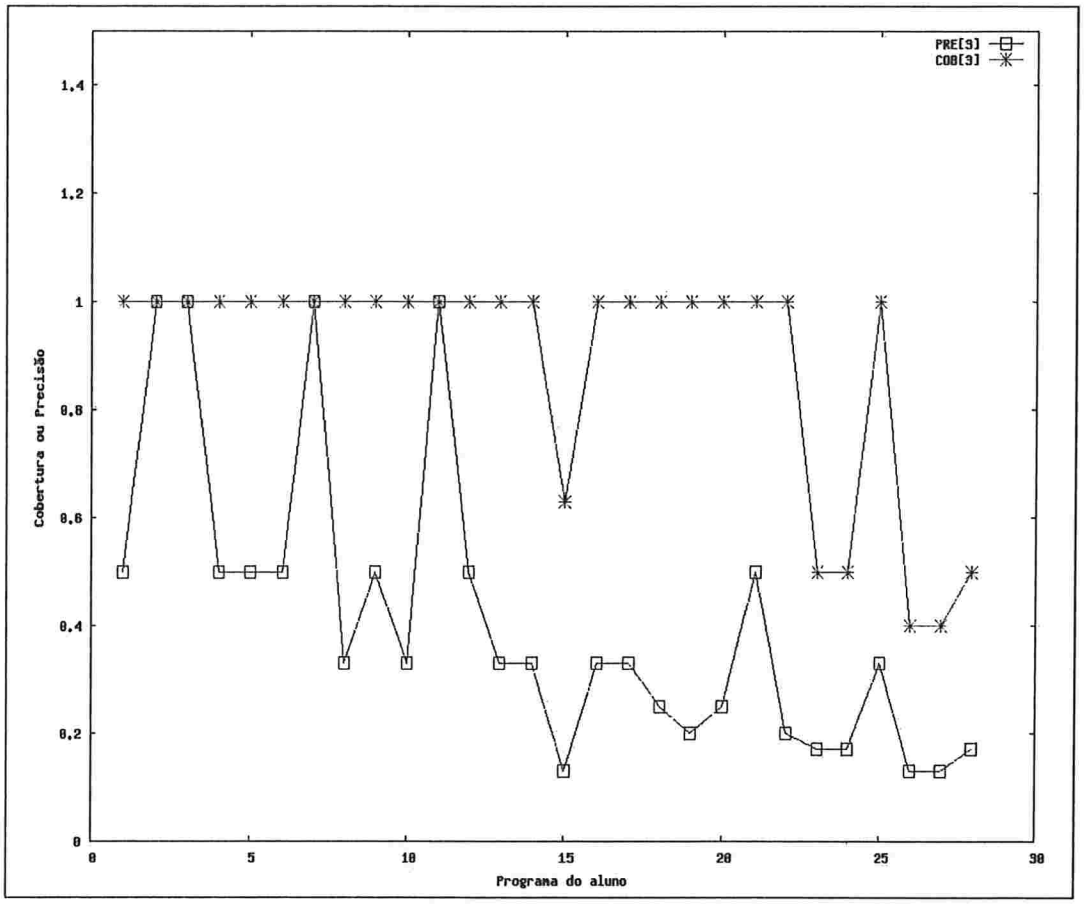


Figura 8.5: PRE e COB para k=3.

Esse exemplo aponta um aspecto interessante do diagnóstico de programas. A falha inserida foi a da linha 8, mas modificando as linhas 9 e a linha 12, também podemos corrigir o programa. Caso o aluno tente corrigir a hipótese de falha da linha 9 ou da linha 12, o programa só funcionará para cada um dos casos de teste. No entanto, se o aluno corrigir a falha na linha 8, o programa funcionará para todos os casos de teste.

Um outro aspecto importante que deve ser observado é com relação ao cálculo heurístico da estimativa de uma sentença estar correta. Embora o processo de diagnóstico seja baseado em um caso de teste, a heurística usada incorpora a informação dos outros casos de teste para ordenar as hipóteses.

Apesar dos resultados satisfatórios, o conjunto de programas avaliados ainda é pequeno o que faz com que as medidas de desempenho sejam pouco informativas. De uma forma geral, os experimentos realizados

mostraram um grande campo de investigação, como será discutido no próximo capítulo.

8.3 Considerações finais

Nesse capítulo foi avaliado o desempenho do sistema PROPAT_DEBUG com relação à sua capacidade em devolver um conjunto pequeno de hipóteses que contenha a falha do aluno. Para isso foram selecionados 28 programas com diferentes tipos de falhas do tipo lógica. As medidas usadas foram precisão, cobertura e medida-F, que medem a capacidade do sistema de devolver um conjunto de hipóteses que inclua as falhas reais do programa e que não devolva um número muito grande de hipóteses incorretas.

Os resultados obtidos foram satisfatórios, uma vez que na maioria dos programas testados a medida de cobertura foi alta e o número de hipóteses geradas foi pequeno.

CAPÍTULO 9

Conclusões e Trabalhos Futuros

Esse trabalho apresentou o desenvolvimento de um sistema de diagnóstico de programas, denominado **ProPAT_deBUG**. O sistema foi implementado em Java para ser usado em um IDE (*Integrated Development Environment*) para alunos de um curso introdutório de programação, implementado como um plug-in Eclipse (chamado de **ProPAT** - Programação Baseada em Padrões) [dBDMM05] [dBDM04] [DdB04].

Uma das principais dificuldades no desenvolvimento do sistema foi a construção automática do modelo baseado em valor a partir do programa do aluno. Para isso foi construído um parser usando a ferramenta ANTLR que converte um programa escrito em uma linguagem C restrita em uma árvore sintática abstrata (AST). A partir da árvore AST é extraída a especificação estrutural e comportamental do programa do aluno.

O sistema de diagnóstico foi avaliado para 28 programas com diferentes tipos de falhas, segundo uma classificação proposta a partir de vários trabalhos sobre psicologia de programação [JSGE83] [Bon85] [SS86a] [SS86b] [Sol86] [PB96]. Apesar de não ter sido feita uma avaliação em termos de aprendizagem com alunos num curso introdutório, a avaliação das hipóteses devolvidas traz resultados que indicam um aspecto essencial num sistema de diagnóstico de programas: apresentar um número pequeno de hipóteses válidas sobre o programa que contenham a falha original do aluno. A avaliação foi realizada em termos do resultado do diagnóstico, isto é, da precisão e cobertura do conjunto de hipóteses devolvidas pelo sistema.

Os valores da medida-F usada na avaliação do sistema mostraram resultados que fazem um bom compromisso entre a precisão e a cobertura do diagnóstico de programa. Os valores também mostraram que o diagnóstico é mais difícil para falhas lógicas estruturais.

Além do desenvolvimento e avaliação do sistema de diagnóstico, as contribuições desse trabalho são:

1. apresentar um estudo sobre as técnicas de depuração automática de programas, tanto na área de engenharia de software como na área de sistemas tutores inteligentes (ITS);

2. apresentar as principais dificuldades no aprendizado de programação e fazer uma classificação de falhas típicas em programas, que resultou numa classificação mais completa do que as encontradas na literatura;
3. construir um ambiente de programação com padrões, chamado PROPAT que envolveu a seleção e especificação de um conjunto de Padrões Elementares de programação. Nessa ferramenta, enquanto o aluno edita um programa, ele pode acessar e inserir Padrões Elementares no programa com a intenção de satisfazer sub-metas de um dado problema. Após compilar o programa, este é testado e em caso de falha o sistema de diagnóstico é chamado;
4. explicar em detalhes as técnicas clássicas de diagnóstico baseada em modelos, entre elas, o modelo conceitual para a tarefa de diagnóstico extraído do trabalho *Problem Solving Methods for Diagnosis* [Ben93] e o algoritmo de Reiter [Rei87] usado para calcular conjuntos minimais de hipóteses;
5. explicar em detalhes as técnicas de diagnóstico baseada em modelos para programas, entre elas, o uso e construção de modelos baseados em valor dando um exemplo detalhado de diagnóstico para um programa simples (dificilmente encontrado na literatura);
6. mostrar como é possível fazer a discriminação de hipóteses de falha, pedindo para que o aluno faça previsões sobre os valores das variáveis de saída dos componentes que pertencem a uma hipótese de falha;
7. propor, através de um exemplo, uma maneira de comunicar as falhas em programas através de scripts de diálogos construidos com base na documentação de padrões elementares, fazendo-se a suposição que o aluno tenha conhecimento prévio sobre o uso e aplicação dos padrões.

9.1 Trabalhos Futuros

9.1.1 Extensões do ProPAT_deBUG

Algumas extensões devem ser feitas na linguagem C restrita para que o sistema de diagnóstico possa ser usado em uma disciplina completa de Introdução à Computação do curso de Ciência da Computação da Universidade de São Paulo. Com base na especificação do modelo já feita nesse trabalho, acreditamos que essas extensões sejam facilmente implementadas, entre elas:

- o comando for;
- definição e chamada de funções;
- vetores e matrizes;
- tipos reais e caracteres;
- funções da biblioteca;

- uso de ponteiros;

Além disso, em trabalhos futuros pretende-se construir um parser de programas na linguagem Java, uma vez que esta tem sido amplamente usada em cursos de introdução à programação.

Um outra extensão deve ser feita com relação aos scripts de diálogos usados para a discriminação de hipóteses de falhas: uma vez que cada componente do modelo corresponde a um padrão elementar, é preciso construir um script para cada padrão candidato a falha.

9.1.2 Modelos hierárquicos para diagnóstico

Para tornar a tarefa de diagnóstico mais tratável é possível empregar um modelo hierárquico do programa, que pode ser gerado automaticamente a partir de fragmentos de modelos hierárquicos. O diagnóstico hierárquico, além de tornar o processo de diagnóstico mais eficiente, pode também alcançar objetivos educacionais, por exemplo, melhorar a comunicação com o aluno através de diálogos de mais alto nível de abstração [dKBBW00].

É interessante observar que o processo de diagnóstico de programas pode ser ao mesmo tempo reparador: uma pergunta feita ao aluno pode fazer com que o aluno "descubra" um ou mais erros que ele cometeu em seu programa. Assim, a comunicação com o aluno durante o diagnóstico pode, a qualquer momento, terminar o processo de discriminação de hipóteses (o aluno ao descobrir seu erro, modifica o programa obtendo um programa sem falhas).

Modelos hierárquicos podem ser construídos de forma *button-up*, a partir de componentes primitivos (componentes que não podem ser decompostos). Chamaremos esse processo de **agregação**. Por outro lado, modelos hierárquicos são sempre usados no diagnóstico de forma *top-down*, i.e., a busca por um componente falho parte de um modelo de alto nível da hierarquia para depois decompô-lo em níveis mais baixos, até atingir componentes primitivos (expressões ou sentenças da linguagem, no caso de modelos de programas).

Agregação. O sistema de diagnóstico de programas **ProPAT_deBUG** que foi descrito neste trabalho, gera a partir do programa do aluno um modelo composto por componentes primitivos (expressões, condicionais, atribuições, repetições, etc.). Porém, componentes que correspondem à estrutura de uma linguagem de programação podem ser agregados. Por exemplo, no programa diagnosticado no Capítulo 7, o componente C3 poderia ser considerado um único componente que corresponde à agregação dos componentes C4, C5, C6, C8 e C9. Esse tipo de agregação segue a própria definição da gramática da linguagem. Uma outra maneira de se construir modelos agregados pode ser feita identificando pedaços de programas que atingem determinados objetivos de programação. Nossa proposta é identificar os padrões elementares empregados pelo próprio aluno através da IDE de programação com padrões **ProPAT**. Assim, o modelo hierárquico será construído com objetivos educacionais: é feita a suposição que através de modelos hierárquicos baseados em padrões elementares, será possível estabelecer a comunicação com o aluno, durante a discriminação de hipóteses de falhas, em

diferentes níveis de abstração.

Decomposição. Com modelos hierárquicos, o diagnóstico do programa pode ser realizado exatamente como proposto nesse trabalho porém, usando um modelo com componentes de alto nível. No caso em que o aluno queira investigar uma hipótese de falha em mais detalhes, o sistema pode decompor os componentes em outros de mais baixo nível, através de uma nova chamada ao sistema de diagnóstico considerando um novo conjunto de componentes. Por exemplo, o aluno pode descobrir que a saída de um componente, que corresponde a um padrão elementar, está incorreta e querer saber, em mais detalhes, em que ponto do padrão o erro foi cometido. Em uma outra situação, o aluno pode descobrir que usou a variável errada (saída de um padrão) para imprimir um resultado e não querer investigar o padrão em mais detalhes.

Padrões Elementares como componentes no modelo do programa

Uma das propostas deste trabalho é a de modelar um padrão como um componente com suas entradas e saídas em um modelo baseado em valor. Desta forma, na construção do modelo do programa como foi descrito na Seção 6.3, deve ser dada a informação de qual agregação de componentes correspondem a um padrão elementar. O modelo dos padrões servirá para identificar outras intenções do aluno além dos valores de variáveis. O diagnóstico hierárquico visto dessa maneira deverá ser capaz de detectar falhas tanto no nível de padrões elementares, como no nível de sentenças.

Um exemplo clássico dessa idéia é dado em [JS84] para resolver o seguinte problema:

"Calcular a média dos elementos de uma seqüencia dada, terminada com 9999"

A Figura 9.1 mostra como as linhas do programa em C podem ser agregadas para a identificação dos seguintes padrões (planos de programação de acordo com [JS84]): *Repetição com Sentinela*, *Divisão*, *Impressão*, *Resultado Válido*.

Identificação de Padrões Elementares no programa do aluno

Para identificar padrões no programa do aluno no ambiente de programação **ProPAT** podemos adotar os seguintes modos de programação do aluno:

- *MODO 1: programação muito controlada*: as sub-metas dos problemas são especificadas pelo professor e o aluno deve selecionar um padrão para resolver cada sub-meta do problema. Neste modo, o aluno só pode programar através da inserção de padrões contidos na biblioteca de padrões do ProPAT;

- *MODO 2: programação pouco controlada*: neste modo o aluno pode, além de inserir padrões, programar livremente. No entanto, o próprio aluno ainda deve indicar quais são as linhas do programa e/ou padrões que satisfazem as sub-metas do problema;

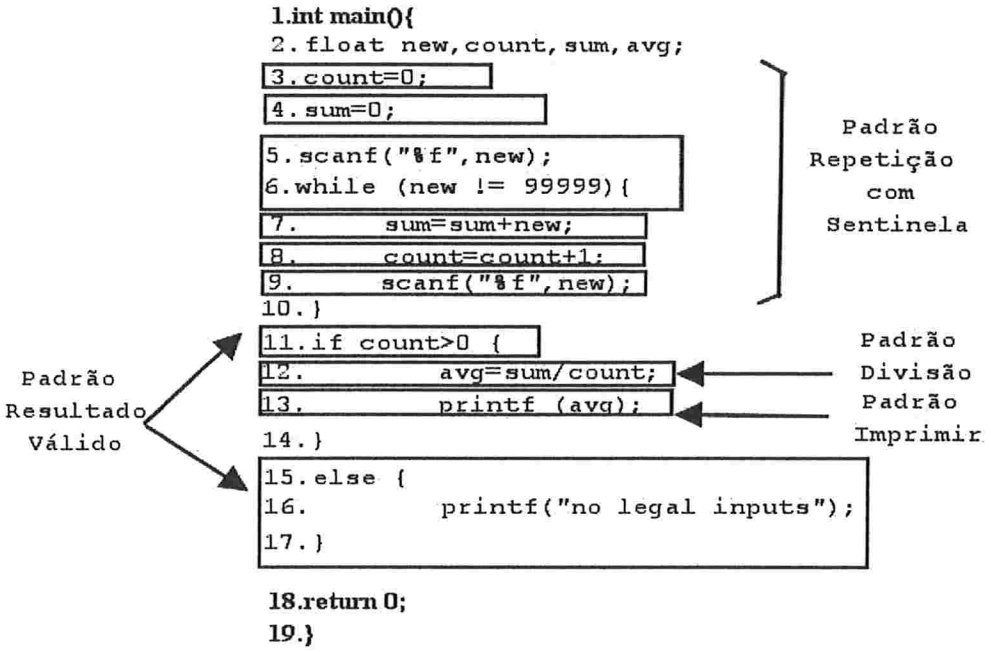


Figura 9.1: Padrões usados para o problema 1

- *MODO 3: programação livre*: o aluno programa livremente com ou sem a inserção automática de padrões e o sistema de diagnóstico deve ser capaz de reconhecer quais foram os padrões usados pelo aluno para atingir as metas do problema.

9.1.3 Ontologia de Padrões Elementares

O modelo para documentação de padrões elementares proposto no Capítulo 4 inclue um conjunto de metadados. Como um trabalho futuro propomos usar os metadados para a construção de uma ontologia sobre programação, com base nos documentos de padrões elementares, para ser usada em tutores inteligentes e ferramentas do tipo LMS (*Learning Management System*).

Referências Bibliográficas

[AC01] ACM/IEEE-CS. Computing curricula 2001, 2001. http://www.sigcse.org/cc2001/. 8

[AJ80] A. Adam and J.Laurent. Understanding and Debugging Novice Programs. *Artif. Intell.*, 15(1,2):75–122, 1980. 23

[App00] Brad Appleton. Patterns and Software: Essential Concepts and Terminology, 2000. http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html. 30

[AS86] J. R. Anderson and E. Skwarecki. The automated tutoring of introductory computer programming. In *16 Communications of the ACM*, pages 842–849, 1986. http://act-r.psy.cmu.edu/publications/pubinfo?id=114. 23, 44

[AW98] Owen Astrachan and Eugene Wallingford. Loop Patterns, 1998. http://www.cs.duke.edu/~ola/patterns/plopd/loops.html. 30

[Ben93] Richard Benjamins. *Problem Solving Methods for Diagnosis.* PhD thesis, University of Amsterdam, 1993. 3, 45, 46, 47, 49, 59, 61, 79, 80, 114

[Ber99] Joseph Bergin. Patterns for Selection Version 4, 1999. http://csis.pace.edu/~bergin/patterns/Patternsv4.html. 30

[Bon85] Jeffrey Bonar. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction*, 1(2):133–161, 1985. 13, 113

[Bri02] Stina Bridgeman. Intro to Computing I, 2002. http://cs.colgate.edu/faculty/stina/courses/cosc/101/f02/syllabus.html. 30, 32

[CR04] Luca Chittaro and Roberto Ranon. Hierarchical model-based diagnosis based on structural abstraction. *Artif. Intell.*, 155(1-2):147–182, 2004. 59

[CW03] Rong Chen and Franz Wotawa. Debugging with an Enriched Dependency-based Model or How to Distinguish between Aliasing and Value Assignment. In *Seventeenth International Workshop on Qualitative Reasoning*, 2003. 63

[Dav84] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artif. Intell.*, 24(1-3):347–410, 1984. 48, 59

[dBDM04] Leliane Nunes de Barros, Karina Valdivia Delgado, and A.G. Machion. An ITS for Programming to explore Practical Reasoning. In *Simpósio Brasileiro de Informática na Educação SBIE*, 2004. 113

[dBDMM05] Leliane Nunes de Barros, Karina Valdivia Delgado, Ana Mota, and Patricia Matsumoto. A Tool for Programming Learning with Pedagogical Patterns. In *Eclipse Technology eXchange eTX at OOPSLA*, 2005. 113

[DdB04] Karina Valdivia Delgado and Leliane Nunes de Barros. ProPAT: A Programming ITS Based on Pedagogical Patterns. In *Seventh International Conference on Intelligent Tutoring Systems*, volume 3220, pages 812–814. Springer-Verlag Lecture Notes in Computer Science, 2004. 113

[de 86] J de Kleer. Problem solving with the ATMS. *Artif. Intell.*, 28(1):197–224, 1986. 52

[dKBBW00] K. de Koning, B. Bredeweg, J. Breuker, and B. Wielinga. Model-Based Reasoning about Learner Behaviour. *Artificial Intelligence*, 117(2):173–229, 2000. 115

[Duc93] M. Ducasse. A Pragmatic Survey of Automated Debugging. In *Proc. 1st Workshop on Automated and Algorithmic Debugging*, volume 749 of *LNCS*, 1993. `http://citeseer.ist.psu.edu/367030.html`. 19, 20

[dW76] J de Kleer and B C Williams. Local Methods of Localizing Faults in Electronic Circuits. *MIT. AI Lab Memo*, 394, 1976. 45

[dW87] J de Kleer and B C Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987. 48, 50, 57, 79

[DZ01] Adair Dingle and Carol Zander. Assessing the ripple effect of CS1 language choice. *J. Comput. Small Coll.*, 16(2):85–93, 2001. 8

[Ecl] Eclipse. Eclipse.org. `http://www.eclipse.org/`. 37

[ETW+96] J. Philip East, Rebecca Thomas, Eugene Wallingford, Walter Beck, and Janet Drake. Pattern-Based Programming Instruction. In *the Proceedings of the ASEE Annual Conference and Exposition, Washington, DC*, June 1996. 29

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987. 11

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Adisson Wesley, 1995. 31

[Gro] Hillside Group. Patterns. `http://hillside.net/patterns/definition.html`. 30

[GSW89] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A Correction to the Algorithm in Reiter's Theory of Diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989. xix, 51, 53

[IEE90] IEEE Standard Glossary of Software Engineering Terminology. Technical Report ANSI/IEEE Std 610.12, 1990. 9

[JHS02] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization, 2002. xx, 81, 82

[Joh90] W. Lewis Johnson. Understanding and Debugging Novice Programs. *Artif. Intell.*, 42(1):51–97, 1990. 23, 24, 29

[JS84] W. Lewis Johnson and Elliot Soloway. Proust: Knowledge-based program understanding. In *Proceedings of the 7th international conference on Software engineering, Florida, United States*, pages 369 – 380, 1984. 3, 23, 24, 44, 116

[JSGE83] Saj-Nicole Joni, Elliot Soloway, Robert Goldman, and Kate Ehrlich. Just so stories: how the program got that bug. *SIGCUE Outlook*, 17(4):13–26, 1983. 13, 113

[Kum02] A. Kumar. Model-Based Reasoning for Domain Modeling in a Web-Based Intelligent Tutoring System to Help Students Learn to Debug C++. In *Proceedings of Intelligent Tutoring Systems, LNCS 2363,France*, pages 792–801, 2002. 23, 44

[LV03] H. Chad Lane and Kurt VanLehn. Coached program planning: dialogue-based support for novice program design. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 148–152. ACM Press, 2003. 25

[LV04] H. Chad Lane and Kurt VanLehn. A Dialogue-based Tutoring System for Beginning Programming. In *Proceedings of the 17th International Florida Artificial Intelligence Research Symposium (FLAIRS)*, 2004. 25

[Moz92] Igor Mozetic. Model-Based Diagnosis: An Overview. In *Advanced Topics in Artificial Intelligence*, pages 419–430, 1992. 45

[MS+02] Wolfgang Mayer, , Markus Stumptner, Dominik Wieland, and Franz Wotawa. Observations and Results Gained from the Jade Project. In *International Workshop on Principles of Diagnosis DX-02*, 2002. 3, 61, 82

[MSW00] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. A Value-Based Diagnosis Model for Java Programs. In *Eleventh International Workshop on Principles of Diagnosis (DX)*, 2000. `http://www.dbai.tuwien.ac.at/staff/wotawa/dx2000c.ps.gz`. 1, 3, 4, 62, 64, 66

[MSWW00] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. JAVA AI Support for Debugging Java Programs, 2000. `http://citeseer.ist.psu.edu/383111.html`. 62

[MSWW02] Wolfgang Mayer, Markus Stumptner, Dominik Wieland, and Franz Wotawa. Can AI help to improve debugging substantially? Debugging Experiences with Value-Based Models. In *Proc. ECAI*, pages 417–421, Lyon, 2002. 72

[Mur89] William R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989. 23

[Par89] Terence Parr. ANTLR: Parser generator, 1989. `http://www.antlr.org/`. 76

[Pat] Pedagogical Patterns. The Pedagogical Patterns Project. `http://www.pedagogicalpatterns.org/`. 30

[PB96] J.F. Pane and B.A.Myers. Usability Issues in the Design of Novice Programming Systems. *Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-96-132*, 1996. 13, 113

[PC03] Ron Porter and Paul Calder. A pattern-based problem-solving process for novice programmers. In *Proceedings of the fifth Australian Conference on Computing education*, pages 231–238. Australian Computer Society, Inc., 2003. 30

[Pi00] CDT Plug-in. The eclipse CDT plug-in, 2000. `http://www.eclipse.org/cdt/`. 37

[Pro00] Viera K. Proulx. Programming patterns and design patterns in the introductory computer science course. In *Proceedings of the thirty-first SIGCSE Technical Symposium on Computer Science Education*, pages 80–84. ACM Press, 2000. 3, 30

[Pro04] ProPAT. The proPAT plug-in, 2004. `http://www.ime.usp.br/~articuno/eclipse/`. 37

[Rei87] R Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987. xix, 48, 49, 50, 51, 52, 53, 114

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition).* Prentice Hall, 2003. 77

[Sel93] Jonh Self. Model-Based Cognitive Diagnosis. In *AAI, AI-ED Technical Report No 82*, 1993. `http://citeseer.nj.nec.com/self93modelbased.html`. 1

[SGD04] Goran Simic, Dragan Gasevic, and Vladan Devedzic. Semantic Web and Intelligent Learning Management Systems. In *Proc. ITS 2004 Workshop on Applications of Semantic Web Technologies for Web-based ITS*, 2004. 26, 32

[Sol86] Elliot Soloway. Learning to program = learning to construct mechanism and explanations special section, 1986. 13, 113

[SS86a] J. C. Spohrer and E. Soloway. Alternatives to construct-based programming misconceptions. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 183–191, New York, NY, USA, 1986. ACM Press. 13, 113

[SS86b] James C. Spohrer and Elliot Soloway. Novice mistakes: are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, 1986. 13, 113

[SW98] Markus Stumptner and Franz Wotawa. A Survey of Intelligent Debugging. *AI Communications 11*, pages 35–51, 1998. `http://citeseer.nj.nec.com/stumptner98survey.html`. 21, 23

[VR79] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition.* Dept. of Computer Science, University of Glasgow, 1979. 100

[Wal01] Eugene Wallingford. The Elementary Patterns Home Page, 2001. `http://www.cs.uni.edu/~wallingf/patterns/elementary/`. 30

[Was99] R. Wassermann. *Resource-Bounded Belief Revision.* PhD thesis, Institute for Logic, Language and Computation of the University of Amsterdam, 1999. 52

[Wen87] Etienne Wenger. *Artificial intelligence and tutoring systems: computational and cognitive approaches to the communication of knowledge.* Morgan Kaufmann Publishers Inc., 1987. 1, 22

[Wie01] Dominik Wieland. *Model-Based Debugging of Java Programs Using Dependencies.* PhD thesis, Technische Universität Wien, 2001. 8, 9, 20, 62, 63

[Win96] Leon E. Winslow. Programming Pedagogy: a Psychological Overview. *SIGCSE Bull.*, 28(3):17–22, 1996. 3, 7