Diagnóstico Baseado em Modelos num Sistema Tutor Inteligente para Programação com Padrões Pedagógicos

Karina Valdivia Delgado
kvd@ime.usp.br

MONOGRAFIA
DISCIPLINA MAC 5701: TÓPICOS EM COMPUTAÇÃO

Curso: **Mestrado em Ciência da Computação**
Área de Concentração: **Inteligência Artificial**
Orientadora: **Prof. Dra. Leliane Nunes de Barros**

O aluno recebe apoio financiero da CAPES.
- Junho de 2004 -

Lista de Figuras

Conteúdo

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 Inteligente de Programação é a depuração do programa construído pelo aluno. Dentre as propostas de depuração automática de programas, a técnica de IA denominada Model Based Diagnosis (MBD), tem apresentado bons resultados para diagnosticar programas escritos por programadores experientes. MBD analisa um modelo do programa representado na forma de componentes e conexões, onde os componentes correspondem às estruturas lógicas da linguagem de programação. No entanto, todas as propostas de depuração automática são consideradas insuficientes por não detectarem todos os erros não-sintáticos, isto é, erros que estão relacionados à intenção do programador e sua implementação no código.

Este artigo propõe a implementação de um sistema de diagnóstico do tipo MBD para a analisar programas de alunos em um sistema tutor inteligente.

1 Introdução

O diagnóstico cognitivo é o processo de inferir o estado cognitivo de uma pessoa a partir de seu desempenho. Para os Sistemas Tutores Inteligentes (*Intelligent Tutoring Systems ITS*), o termo *diagnóstico cognitivo* é usado freqüentemente como sinônimo de *modelagem do estudante* para denotar o processo de construir uma representação do conhecimento do aluno a partir das evidências fornecidas pelas soluções de problemas [Self, 1993][1].

Em um Sistema Tutor Inteligente de Programação (*Intelligent Programming Tutor IPT*), o diagnóstico automático do programa do estudante é essencial e é considerada uma tarefa difícil pois podem existir muitas variações num programa que satisfazem as exigências especificadas do problema proposto ao aluno. Não há ainda uma ferramenta eficiente para detectar erros semânticos e lógicos de um programa (como o compilador que detecta erros de sintaxe). Além disso, espera-se que o estudante tenha algumas oportunidades de aprendizagem durante o processo de diagnóstico de seu programa, dependendo da maneira como seus erros são detectados e comunicados pelo sistema.

Dentre as propostas de depuração automática de programas, a técnica de IA denominada Model Based Diagnosis (MBD), tem apresentado bons resultados para diagnosticar programas escritos por programadores experientes. MBD analisa um modelo do programa representado na forma de componentes e conexões, onde os componentes correspondem às estruturas lógicas da linguagem de programação. Nesta proposta, o programador deve fazer previsões sobre os valores das variáveis para serem comparadas com valores observados no programa. No entanto, todas as propostas de depuração automática são consideradas insuficientes por não detectarem todos os erros não-sintáticos, isto é, erros que estão relacionados à intenção do programador e sua implementação no código.

[1] 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 [Wenger, 1987].

Para identificar a intenção do programador, é preciso compreender como eles resolvem problemas de programação. Pesquisas em teorias cognitivas 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 que aprenderam. Incentivados por essas idéias, educadores de programação desenvolveram um modelo baseado em padrões para ensino de programação. Neste modelo, o aprendizado pode ser visto como um processo de reconhecimento de padrões que compara soluções gerais aprendidas com a situação atual de resolução de problemas.

Neste trabalho, apresentamos um ambiente de programação em que o estudante pode programar usando um conjunto de padrões pedagógicos, isto é, padrões de programação recomendados por um grupo de educadores. Neste ambiente, enquanto o aluno edita um programa, ele pode selecionar e inserir padrões no programa com a intenção de satisfazer sub-metas de um dado problema. Após o programa ser compilado com sucesso, este será submetido ao sistema de diagnóstico (MBD) a fim de detectar possíveis erros no programa e/ou erros conceituais do aluno.

Uma visão geral das técnicas de depuração automática são dadas na Seção 3. Na Seção seguinte são mostrados alguns Tutores Inteligentes de Programação. Na Seção 2, apresentamos alguns exemplos de padrões pedagógicos de programação. Na Seção 6 mostramos a ferramenta ProPAT, um ambiente para aprendizes programarem usando padrões de programação. Na Seção 7, descrevemos como o sistema de diagnóstico baseado em modelos pode ser integrado ao ambiente ProPAT, para diagnosticar (depurar) falhas no programa do aluno de forma interativa.

2 Ensino 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 memorizar a sintaxe e a semântica de uma linguagem de programação
- **Aprender a resolver problemas para o computador executar**: o estudante deve "traduzir"uma solução conhecida (por exemplo, resolver uma equação de segundo grau) para um programa que o computador consiga executar

Uma linguagem de programação possui muitos detalhes, apesar disso dificultar a vida do aprendiz, não parece ser essa a maior dificuldade que ele deve enfrentar, 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, comum às diferentes linguagens.

[Johnson and Soloway, 1984] aponta evidências que sugerem que programadores experientes armazenam e recuperam experiências passadas, ou *padrões* de soluções de problemas, que estejam relacionados ao novo problema e que possam ser adaptadas à situação atual. Por outro lado, um aprendiz não tem nenhuma experiência real de programação em mente e portanto só pode procurar por sentenças da linguagem que aprenderam.

2.1 Escolha da linguagem de programação para ensino

A escolha da linguagem de programação a ser ensinada na disciplina de Introdução à Computação deve ser feita com certos cuidados devido aos efeitos que ela pode ter no desenvolvimento do aluno em suas próximas disciplinas do curriculum, principalmente para aqueles estudantes da área de Ciências de Computação. Os criterios principais para a escolha da linguagem são: as características da linguagem que facilitam o ensino (razões pedagógicas) e linguagens de maior interesse pelo mercado do trabalho [Dingle and Zander, 2001].

Entre as linguagens de programação mais usadas temos as linguagens procedimentais tradicionais como C, Pascal e Modula-2 e as populares linguagens orientadas a objetos como C++ e Java.

Neste trabalho adotou-se a linguagem C++ porém, o diagnóstico de programas pode ser extendido para outras linguagens, de forma que essa proposta independe da linguagem adotada.

3 Depuração automática

Na literatura de depuração automática existem duas categorias diferentes de sistemas: *Tutores Inteligentes de Programação* e *Sistemas de Diagnóstico de Programas*.

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; relacionados a problemas comuns e portanto suas soluções podem ser repetidas. Para esses sistemas é feita a suposição que o aprendizado ocorre na comunicação entre o tutor e o estudante durante o processo de depuração.

Sistemas de Diagnóstico de Programas trabalham com programas escritos por programadores experientes. Os programas são geralmente grandes, originais, complexos e usam linguagens de programação completas e não podem ser completamente especificados. O processo de depuração é executado não com a finalidade de ensinar mas 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 Depuração Automática

Em [Ducasse, 1993] é apresentada uma visão geral de alguns dos sistemas de depuração automática das duas categorias: *Tutores Inteligentes de Programação* e *Sistemas de Diagnóstico de Programas*. Nesse trabalho, são definidas três estratégias de depuração: ***verificação com respeito à especificação***[2], ***verificação com respeito ao conhecimento da linguagem***[3] e ***filtragem com respeito aos sintomas***[4]. Muitos sistemas combinam estas estratégias de acordo com a quantidade de conhecimento disponível.

A técnica de **verificação com respeito à especificação** compara o programa real com uma especificação formal existente. A técnica de **verificação com respeito ao conhecimento da linguagem**, onde o conhecimento é geralmente com relação às restrições da linguagem (por exemplo em C variáveis devem ser declaradas antes de serem usadas), 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 podem ser aplicadas somente a programas pequenos.

Nos *Sistemas de Diagnóstico de Programa*, ***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 causado o sintoma observado de falha. Além disso, com esta estratégia, no máximo o programa inteiro será analisado ao contrário da estratégia de verificação com respeito ao conhecimento da linguagem onde mais erros podem ser gerados do que o número de linhas do código fonte. Assim, a técnica de filtragem deveria ser usada como uma etapa preliminar na presença de um sintoma de erro. Exemplos de técnicas de filtragem são: ***Depuração Algorítmica*** (*algorithmic debugging*), ***Corte do Programa*** (*program slicing*), ***Mutação do Programa*** (*program mutation*), ***Depuração Probabilística*** (*probabilistic debugging*) e ***Diagnóstico Baseado em Modelos (MBD)***. A figura 1 mostra parte da classificação proposta por [Ducasse, 1993] e [Wieland, 2001], focalizando-se nas técnicas de filtragem, detalhadas a seguir:

- *Depuração Algorítmica*, também conhecida como depuração declarativa, se baseia em encontrar falhas no nível de chamadas a métodos sendo elevada a necessidade de interação com o usuário.
- *Corte do Programa*, poda todas as sentenças que não causam o valor errado ou a seqüência de controle errada. O que resta, chamada também de corte ou fatia, pode não conter o erro, nesse caso esta técnica não garante encontrar a falha.
- A técnica de *Mutação do Programa* modifica partes pequenas do programa e se (o mutante) não elimina a falha, as peças que foram modificadas são supostas corretas. Os mutantes não somente fornecem a posição do erro, mas também podem dar sugestões de reparo.

[2] *verification with respect to specification*
[3] *checking with respect to language knowledge*
[4] *filtering with respect to symptom*

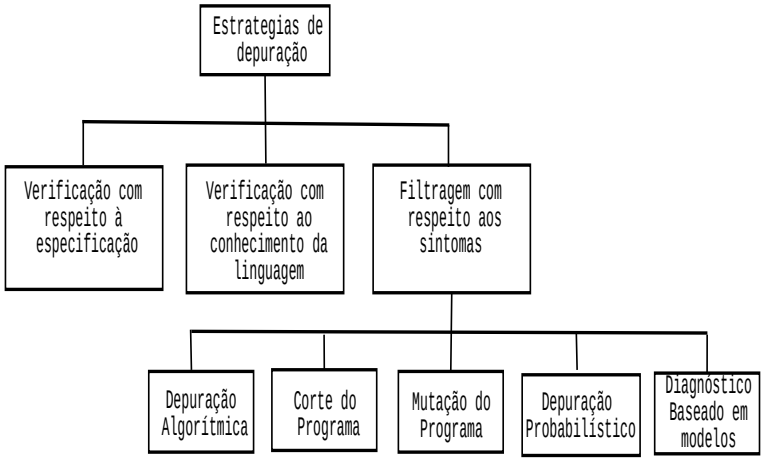


Figura 1: Estratégias de depuração focalizando-se nas técnicas de filtragem

- *Depuração Probabilística* usa probabilidades de falha especificadas por peritos. Computa todas as posições de falha potenciais e usa uma rede de crenças para encontrar as sentenças defeituosas mais prováveis .

- *Diagnóstico Baseado em Modelos* é um tipo de técnica de filtragem que é sucesso para o diagnóstico de sistemas físicos e que também tem sido usada para diagnóstico de programas [Stumptner and Wotawa, 1998] Os primeiros trabalhos se focalizaram na programação lógica, linguagens de projetos de hardware como VHDL e linguagens funcionais. Trabalhos recentes usam linguagens imperativos com execução seqüencial e linguagens orientado a objetos como Java ou C++. Usa uma descrição lógica do sistema de software, composta por componentes e conexões. A descrição é baseada no código do programa e na informação sobre a semântica da linguagem de programação. Eles correspondem ao estado da arte das técnicas de IA para diagnóstico automático. O aspecto central para o sucesso desta estratégia é a construção de modelos eficazes. Os valores de saída desejados devem ser fornecidos pelo estudante e é durante esta comunicação que o ITS deve explorar oportunidades de aprendizagem.

3.2 Técnicas de diagnóstico de Tutores Inteligentes de Programação e Técnicas de Sistemas de Diagnóstico de Programas

Todas as técnicas propostas para *Tutores Inteligentes de Programação* e *Sistemas de Diagnóstico de Programas* são consideradas insuficientes para tratar casos reais. As técnicas de *Tutores de Programação* que podem encontrar erros com precisão, são muito caras em termos computacionais para serem usadas em programas inteiros. Por outro lado, as técnicas de *Diagnóstico de Programas*, consideradas práti-

cas, encontram erros aproximados. Além disso, métodos de *Diagnóstico de Programas* que procuram por erros, meramente inspecionando o código, não podem lidar com uma grande variedade de problemas de lógica de programação. Aqueles métodos não reconhecem que os 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 [Wenger, 1987].

As técnicas de **filtragem** a pesar de ser mais exatas para finalidades de diagnóstico do que outros métodos, ainda não têm claro como podem ajudar aos estudantes a aprenderem com este processo.

4 Tutores Inteligentes de Programação

4.1 Exemplos de Tutores Inteligentes de Programação

Alguns exemplos de sistemas tutores são mostrados na Tabela 1 (em ordem cronológica) com as técnicas empregadas para depuração automática. Observe que o sistema de diagnóstico que estamos propondo, implementa uma nova combinação de abordagens:

- Verificação com respeito ao conhecimento da linguagem: verificação do emprego dos padrões de programação disponíveis pelo professor para resolver o problema dado.
- 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 a padrões de programação.

Descrevemos brevemente o sistema que motivou nossa proposta.

4.2 O Sistema Tutor PROUST

O sistema de diagnóstico proposto para o tutor PROPAT é baseado no tutor **PROUST** [Johnson and Soloway, 1984]: um sistema baseado em intenções. PROUST foi desenvolvido com base na teoria da psicologia do processo de programação. PROUST começa a analisar o programa do estudante a partir de uma *descrição declarativa do problema*, isto é, uma lista dos requisitos a serem satisfeitos, especificados pelo professor.

Soloway propõe um método para determinar os erros no programa que o aluno está tentando fazer, e as idéias erradas que ele poderia ter para explicar a presença dos erros. Esse método envolve dois passos:

1. reconstrução dos objetivos que o estudante está tentando satisfazer e
2. identificação das unidades funcionais no programa que foram usadas para atingir esses objetivos.

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

- o conjunto de objetivos do problema;

Nome	Verificação com respeito à especificação	Verificação com respeito ao conhecimento da linguagem	Outras Filtragem	MBD
INTELLIGENT PROGRAM ANALYSIS	X			
PUDSY	X	X		
LAURA	X			
PHENARETE		X		
PROUST [Johnson and Soloway, 1984]	X	X	X	
TALUS	X			
LISP TUTOR [Anderson and Skwarecki, 1986]	X			
APROPOS	X	X		
KUMAR'S TUTOR [Kumar, 2002]				X
PROPAT		X		X

Tabela 1: Alguns Tutores Inteligentes de Programação mais citados e técnicas que eles implementam para depuração automática. O módulo de diagnóstico de PROPAT proposto implementa uma nova combinação de abordagens: verificação com respeito ao conhecimento da linguagem e MBD.

- a base de conhecimento (biblioteca) de planos de programação, que são estratégias para identificar intenções no código do aluno e
- uma biblioteca de erros comuns.

A tarefa de reconhecimento dos planos do aluno no programa 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 decompõe objetivos de problemas de forma hierárquica através dos planos: planos decompõem objetivos em sub-objetivos ou em 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 (seção de código cujo comportamento não concorda com a especificação do programa) são descobertos com a comparação de planos com o programa do aluno. Programas com erros podem ser derivados de uma decomposição de objetivos incorreta ou de implementações incorretas de decomposições de objetivos corretas. PROUST começa selecionando um objetivo da descrição do problema, recuperando o conjunto de planos da base de conhecimento que

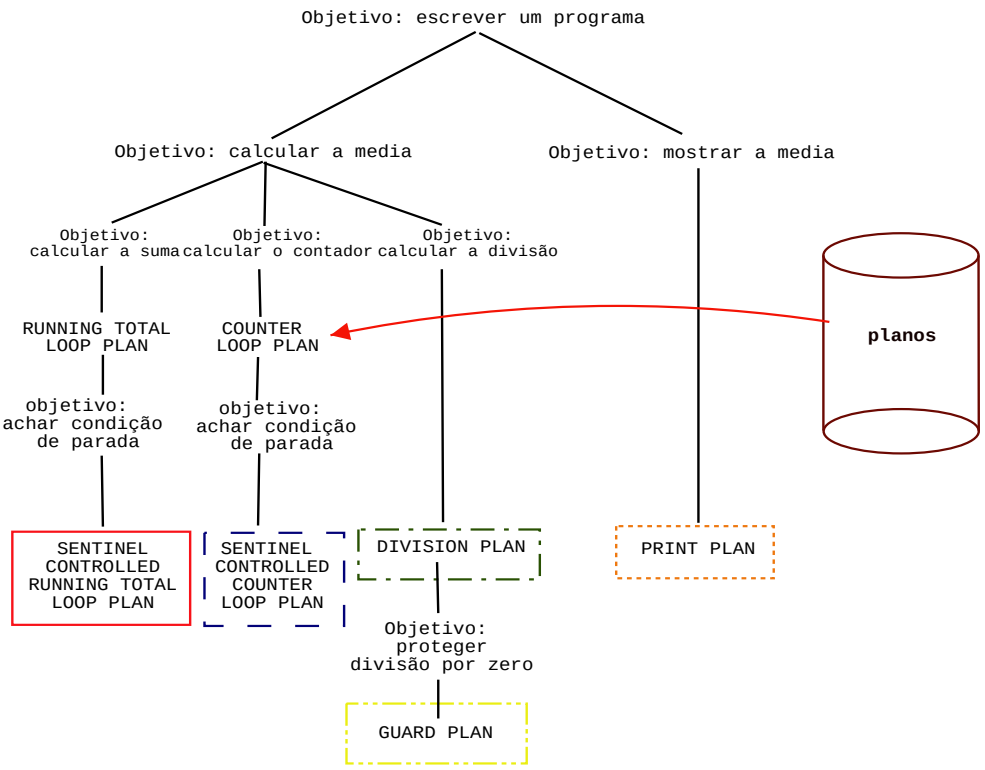


Figura 2: Decomposição de objetivos em sub-objetivos e planos para o problema 1

satisfazem (implementam) o mesmo, para compará-los com o código. Uma vez que planos podem ter sub-objetivos, este processo é recursivo.

O exemplo de problema dado a seguir [Johnson and Soloway, 1984], ilustra de forma clara como PROUST funciona.

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

A Figura 2 mostra a descomposição de objetivos em sub-objetivos e planos para o problema 1, e a Figura 3 mostra como essa decomposição está relacionada ao código feito em C.

4.2.1 Limitações do tutor PROUST

Dentre as limitações de PROUST estão:

- para que o diagnóstico seja correto, é necessário que exista uma boa biblioteca de planos especificadas por educadores experientes de programação, uma vez que aqueles planos devem corresponder às diferentes maneiras que estudantes aprendizes resolvem problemas.

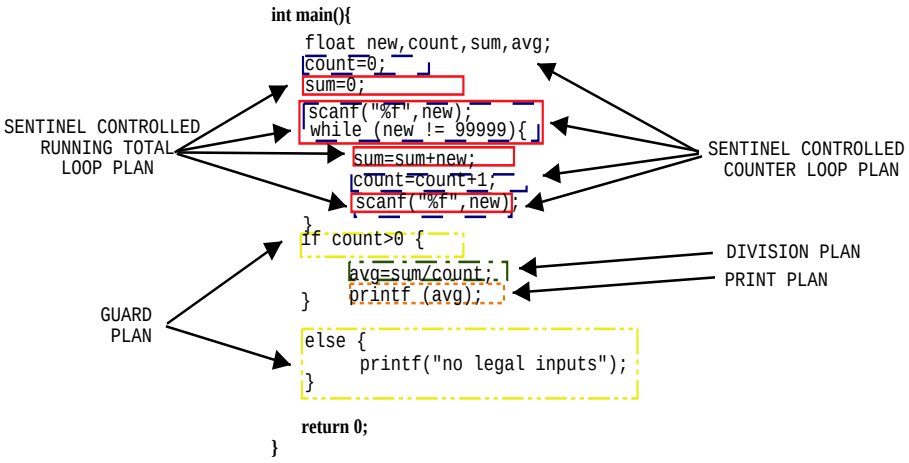


Figura 3: Planos usados para o problema 1

- para muitos problemas, não é possível colecionar um conjunto completo de planos que os solucione;
- professores nem sempre concordam sobre quais planos devem ser incluídos na biblioteca
- professores não podem adicionar novos elementos na biblioteca de planos

5 Ensino de programação baseado em padrões pedagógicos

Incentivados pelas idéias da psicologia de programação 2, foi desenvolvido um modelo baseado em padrões para instrução em programação [East et al., 1996]. Neste modelo, o aprendizado pode ser visto como um processo de reconhecimento de padrões que compara experiências passadas com a situação atual de resolução de problemas.

Desta forma, o enfoque para ensinar programação é apresentar aos estudantes partes pequenas de programas ao invés de esperar que eles escrevam programas inteiros a partir do zero (de um programa vazio). Tais partes de código são chamados **Padrões de Programação Pedagógicos**, que são descritos em termos de uma situação de programação (que poderiam estar relacionadas às intenções de soluções de metas de programação do aluno) em uma linguagem de programação particular (e.g. , Java, C ou C++). Os padrões podem ser ensinados pelo professor ou através de um material impresso ou digitalizado. Padrões são projetados por um professor experiente ou recomendados por um grupo de educadores de programação.

Podemos fazer uma correlação entre os planos propostos pelo PROUST e os padrões de programação. No entanto, os planos no sistema PROUST não possuiam a preocupação de serem didáticos. Além disso,

não era feita a suposição que os planos contidos no sistema deveriam ser ensinados (apresentados) ao aluno pelo professor (ou pela ferramenta).

5.1 Padrões pedagógicos de programação

Os *Padrões Pedagógicos de Programação*, também chamados de *Padrões Elementares de Programação* são descrições de problemas comuns e suas soluções em um formato que, entre outras coisas, facilite o reuso. Os padrões são simples, concisos e seu uso no contexto de aprendizes é recomendado por pesquisadores sobre ensino de programação [Porter and Calder, 2003]. Assim, um padrão associa um problema a uma solução e fornece a informação sobre a situação em que pode ser aplicado. Seu uso potencial no contexto de aprendizagem de programação tem sido explorado pela comunidade de Padrões pedagógicos. Padrões elementares estão disponíveis na web para as linguagens C, C++ e Java [Wallingford, 2001]. Ao longo da década de 90, professores de programação têm documentado padrões que todo aprendiz de programação deve saber, incluindo: padrões de seleção [Bergin, 1999], padrões de repetição [Astrachan and Wallingford, 1998] e outros [Bridgeman, 2002]. Porter e Calder [Porter and Calder, 2003] sugerem um processo para aplicar padrões de programação no ensino em sala de aula e Proulx [Proulx, 2000] criou uma estrutura para um primeiro curso em ciência da computação, baseado em padrões elementares de programação.

Um padrão geral (por exemplo, para projetos de arquitetura, engenharia ou programação) possui quatro elementos essenciais [Gamma et al., 1995]:

- **Nome do padrão**: é a forma que usamos para descrever um problema, suas soluções e conseqüências descritas em poucas palavras. Encontrar nomes bons tem sido uma das partes mais difíceis no projeto de um catálogo de padrões .

- **Problema**: descreve quando aplicar o padrão, explica o problema e seu contexto. Em geral, trata-se de um problema geral que pode ocorrer em vários projetos diferentes. Normalmente o problema inclui uma lista das condições que deveriam ser cumpridas antes de poder aplicar o padrão.

- **Solução**: descreve os elementos usados pelo padrão e seus relacionamentos. Uma recomendação é que a solução não deve descrever um projeto completo dado que o padrão é como um molde que pode ser aplicado em diferentes situações. Fornece uma descrição abstracta de como os elementos resolvem o problema.

- **Conseqüências**: são os resultados e compromissos (*trade-off*s) de se aplicar o padrão. São críticas para a avaliação de alternativas e para ter uma idéia dos custos e benefícios do uso do padrão.

[Gamma et al., 1995] descreve um *design pattern*, mais voltado para projetos de programação orientada a objetos, usando o seguinte formato:

- **Nome do padrão e classificação:** o nome do padrão indica a essência do padrão. Um bom nome é vital, já que será parte do vocabulário do projeto.
- **Intenção (Intent)**, uma sentença curta que responde às seguintes perguntas: o que o padrão faz? qual é a sua intenção? para que problema ele está direcionado?
- **Nomes alternativos**: outros nomes conhecidos do padrão.
- **Motivação**: Um cenário que ilustra o problema e como é resolvido pelo padrão. O cenário ajuda a entender melhor a descrição abstracta do padrão.
- **Aplicabilidade**: quais são as situações em que o padrão pode ser aplicado incluindo exemplos de projetos ruins para os quais o padrão poderia ser empregado melhorando esta situação.
- **Estrutura**: uma representação gráfica do padrão.
- **Participantes e colaborações:** classes e objetos, suas responsabilidades e colaborações.
- **Conseqüências**: como o padrão logra seus objetivos, quais são os compromissos e resultados de usar o padrão.
- **Implementação**: que sugestões ou técnicas deveriam ser conhecidas quando implementamos o padrão, existem aspectos específicos da linguagem.
- **Código exemplo**: fragmentos de código que ilustram como deve implementar o padrão em uma linguagem de programação específico.
- **Usos conhecidos**: exemplos do padrão encontrados em sistemas reais. Deveria incluir pelo menos dos exemplos de domínios diferentes.
- **Padrões relacionados**: quais padrões estão diretamente relacionados, quais são as principais diferenças, com que outros padrões este padrão deveria ser usado.

Padrões de programação elementares podem ajudar o programador iniciante em dois aspectos:

- no aprendizado de estratégias gerais (de mais alto nível de abstração);
- na memorização da linguagem de programação, uma vez que sua documentação contém um programa que é um exemplo de aplicação do padrão.

Além disso, fazendo a suposição que padrões elementares de programação podem ser compreendidos e utilizados por alunos de programação, eles podem ser identificados no programa do aluno pelo professor.

nome do padrão	uso/aplicação	sintaxe

Repetição com sentinela	Você quer repetir um conjunto de ações enquanto uma condição for verdadeira. Em geral, o conjunto de acões está relacionado ao processamento de uma seqüência de elementos ou números. A quantidade de elementos é desconhecida mas o fim da sequência é indicado por um valor sentinela. Os elementos podem ser lidos ou gerados.	<INICIALIZAÇÕES>
		<INICILIZAÇÃO DA VARIÁVEL SENTINELA>
		while (<CONDIÇÃO DA VARIÁVEL SENTINELA>){
		<LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQUÊNCIA >
		<PROCESSAR ELEMENTO>
		<ATUALIZAÇÃO DA VARIÁVEL SENTINELA>
		}

Repetição contada	Você quer repetir un determinado número de vezes um conjunto de ações. Em geral, o conjunto de acões está relacionado ao processamento de uma seqüência de elementos ou números. A quantidade de elementos deve ser conhecida. Os elementos podem ser lidos ou gerados.	<INICIALIZAÇÕES>
		<INICILIZAÇÃO DO CONTADOR>
		while (<CONDIÇÃO DO CONTADOR>){
		<LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQUÊNCIA >
		<PROCESSAR ELEMENTO>
		<ATUALIZAÇÃO DO CONTADOR>
		}

Resultado Válido	Você quer validar una variável quando condições límites ocorrem.	**if** (<CONDIÇÃO>){
		<CONJUNTO DE AÇÕES PARA CONDIÇÃO VERDADEIRA>
		}
		else
		<CONJUNTO DE AÇÕES PARA CONDIÇÃO FALSA>
		}

Figura 4: Exemplos de Padrões de Programação

Isso pode permitir que: (1) o professor identifique as intenções do aluno; (2)seja estabelecida uma comunicação melhor entre professor e aluno, já que eles fornecem um vocabulário sobre estratégias gerais de solução de problemas que ambos, professor e aluno, podem adotar.

Alguns exemplos de padrões são mostrados na Figura 4 e na Figura 5 mostramos um deles, *Repetição com Sentinela*, em maior detalhe. Note que numa documentação completa de um padrão, como a apresentada para os *design patterns*, incluem outros aspectos, entre eles, *estrutura*, *intenção*, *nomes alternativos*, *participantes*, *colaboradores*, *conseqüências*, etc., que podem ser melhor exploradas pela comunidade de padrões pedagógicos.

5.2 Padrões e Domínios de aplicação

Na USP a disciplina de Introdução à Computação é dada para vários cursos de áreas diferentes: Ciências Exatas e Tecnologia, Ciências Humanas e Biologicas e para Engenharia. Muitos educadores acreditam que o domínio de aplicação, isto é, os problemas de programação propostos pelo professor, deveriam ser diferentes para cada uma dessas áreas.

Desta forma, acreditamos que padrões elementares devem ser sugeridos para diferentes domínios de aplicação. Por exemplo, para ensinar Introdução à Computação para estudantes de engenharia, seria mais adequado propor problemas de controle de robôs para introduzir laços, funções e classes. Os padrões,

nome do padrão *Repetição com sentinela*

uso/aplicação Você quer repetir um conjunto de ações enquanto uma condição for verdadeira. Em geral, o conjunto de acões está relacionado ao processamento de uma seqüência de elementos ou números. A quantidade de elementos é desconhecida mas o fim da sequência é indicado por um valor sentinela. Os elementos podem ser lidos ou gerados.

padrões dependentes *Declaração de Variável Inteira, Conjunto de Ações*

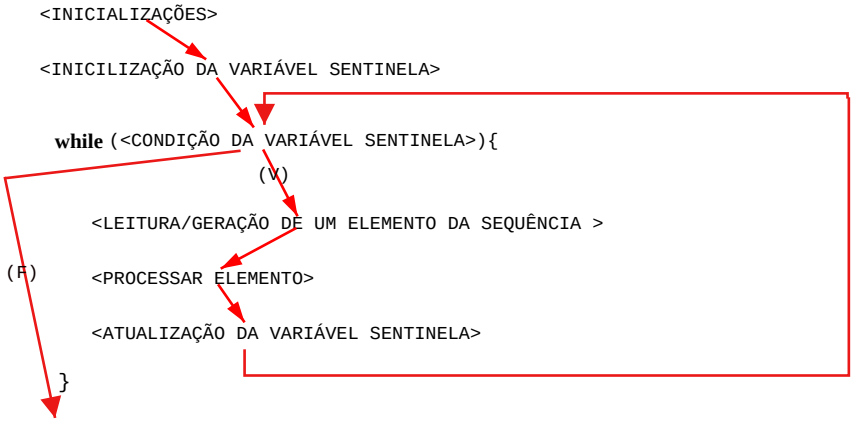
sintaxe

```
<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>

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 seguência 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. Para isso, 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".



exemplo de uso 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*/
}
```

Figura 5: Repetição com sentinela

apesar de serem descrições de problemas comuns, têm uma documentação extensa que inclui por exemplo: aplicabilidade, código de um problema exemplo, usos conhecidos, entre outros. A princípio essa documentação deveria ser descrita de maneira diferente para cada domínio. É importante que a escolha correta de tipos de problemas de exemplo na documentação do padrão, estejam relacionados ao domínio de aplicação, i.e. que estejam de acordo com os interesses do aluno. Por exemplo, no domínio de controle de robôs, o padrão repetição contada da figura 4 o uso/aplicação poderia ser: "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".

6 ProPAT

ProPAT é um sub-projeto do projeto IBM Eclipse em parceria com o Instituto de Matemática e Estatística da Universidade de São Paulo para a construção de um *Ambiente Integrado de Desenvolvimento* (IDE) para cursos de introdução à programação (concluído em março de 2004 http://www.ime.usp.br/ articuno/eclipse/). A idéia básica é construir um plug-in Eclipse que permita ao aluno visualizar, selecionar e programar através de padrões de programação. Além disso, o professor pode inserir seus próprios padrões e exercícios.

Em termos gerais, um plug-in Eclipse é composto de perspectivas e *views*. Esse projeto foi desenvolvido para a linguagem C e está sendo modificado para C++. Algumas das características do ProPAT para C foram herdadas do próprio Eclipse e outras foram desenvolvidas especialmente para o projeto, listadas a seguir (Figura 6):

- **Perspectiva do aluno**: onde os alunos podem escolher exercícios e desenvolver soluções através da seleção de padrões, ou ainda podem escrever livremente seu próprio código.
- **Perspectiva do professor**: usada pelo professor para especificar novos exercícios e padrões, que estarão disponíveis posteriormente para o estudante.
- **Editor *View***: editor de programas no qual o aluno pode incluir padrões automaticamente, substituindo metadados da descrição do padrão (palavras entre apóstrofes) ou inserindo código livremente.
- **Navigator *View***: permite navegar entre projetos e os respectivos arquivos.
- **Pattern *View***: mostra a lista de padrões que podem ser selecionados pelo aluno para (1) visualizar sua documentação (na Pattern Info *View*) ou para (2) inseri-los (no Editor *View*).
- **Pattern Info *View***: mostra a documentação do padrão selecionado.

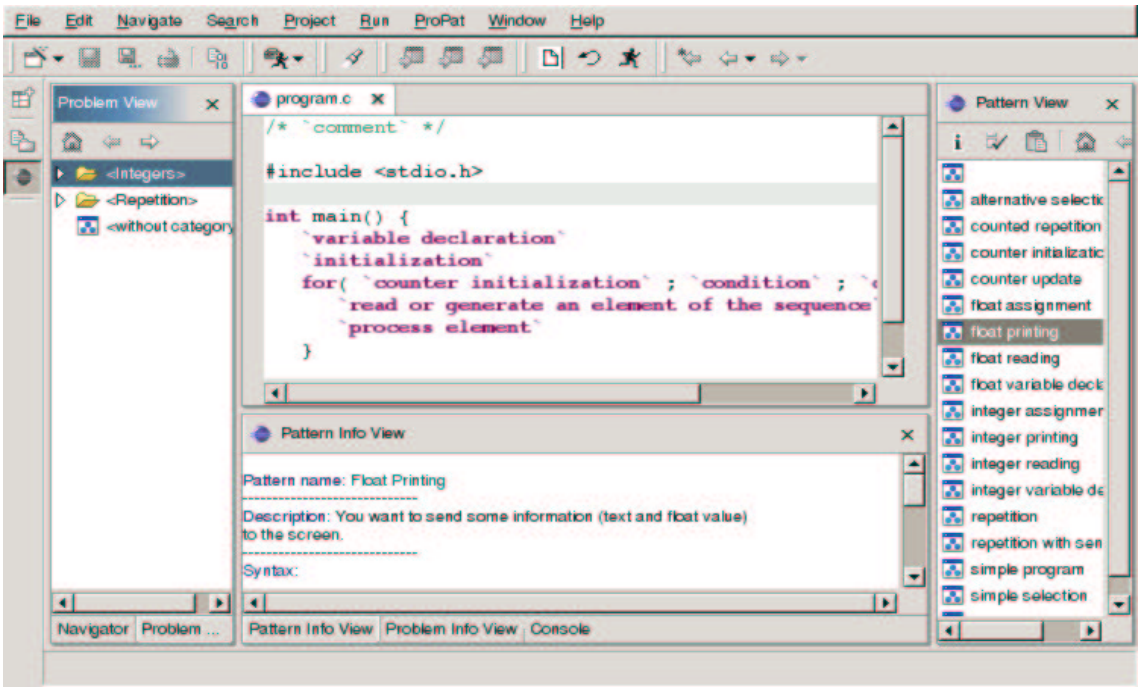


Figura 6: Perspectiva do aluno de ProPAT

- **Problem *View***: permite navegar através da lista de exercícios, agrupados por categorias.
- **Problem Description *View***: mostra a descrição do exercício selecionado.
- **Message Console**: mostra erros de compilação e o resultado da execução do programa.

Neste trabalho, propomos integrar ao plug-in ProPAT um sistema de diagnóstico de programas capaz de se comunicar com o estudante durante o processo de detecção de erros. Desta forma, pretendemos que o ProPAT tenha características de um Sistema Tutor Inteligente (ITS) para Programação. Um ITS é um sistema de ensino/aprendizagem que emprega técnicas de Inteligência Artificial (IA) com o objetivo de promover o aprendizado individualizado. Além disso, um ITS deve ser capaz de fazer planejamento curricular e de planejar sua comunicação com o aluno [Wenger, 1987].

Um dos aspectos centrais de um sistema **Tutor Inteligente de Programação** é o diagnóstico da solução do aluno. Desta forma, o principal objtivo deste trabalho será o sistema de diagnóstico de programas. Mostraremos como técnicas conhecidas de diagnóstico baseado em modelos para sistemas físicos [Benjamins, 1993] podem ser aplicadas para analisar programas [Cristinel Mateis and Wotawa, 2000] e finalmente propomos uma maneira de estender essas técnicas para tornar possível o diagnóstico com o uso de padrões para identificar a estratégia (intenção) da solução do aluno.

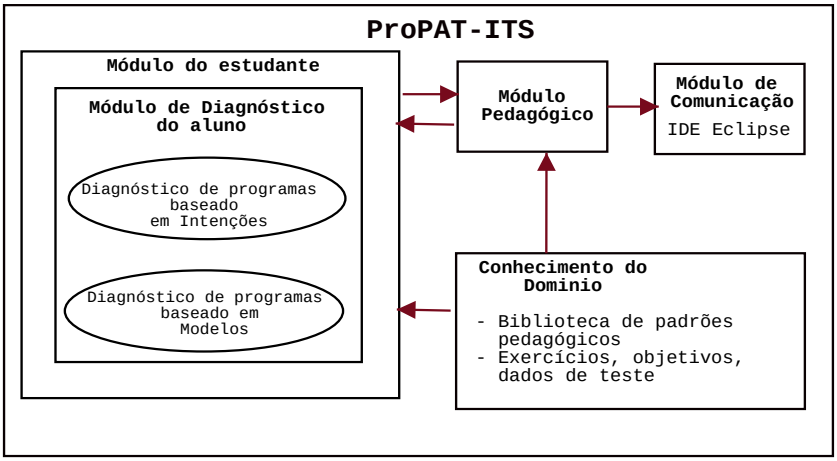


Figura 7: Visão geral do ProPAT como um Sistema Tutor Inteligente.

Na Figura 7 mostramos como o sistema de diagnóstico e o plug-in Eclipse farão parte do sistema tutor para programação que está sendo desenvolvido no IME-USP. Este sistema será composto pelos seguintes módulos: (1) o Módulo do Estudante que representa o estado do aluno, sendo uma parte importante o Módulo de Diagnóstico do aluno foco deste trabalho; (2) o Módulo Pedagógico, que especifica o processo de ensino, quando e como apresentar um novo tópico (que está sendo desenvolvido como um trabalho de doutorado); (3) o Módulo de Conhecimento do Domínio que contém a biblioteca de padrões pedagógicos, exercícios, objetivos e dados de teste (já inseridos no plug-in Eclipse); e (4) o Módulo de Comunicação que é encarregado pelas interações entre o estudante e o sistema (através da IDE de Eclipse).

7 Diagnóstico Baseado em Modelos

Existem duas abordagens para diagnóstico em Inteligência Artificial [Mozetic, 1992]:

- *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 é na forma de sintoma->falha. Em consequência, estes sistemas de diagnóstico são altamente especializados e restritos a aplicações que requerem experiência no domínio, acarretando num custo elevado para desenvolvê-los, mantê-los e estendê-los.

- *Diagnóstico Baseado em modelos*, chamado também de *Diagnóstico Baseado em Princípios Básicos (Diagnosis from the First Principles)*. O primeiro sistema MBD foi desenvolvido por de Kleer em 1976 [de Kleer and Williams, 1976]. O paradigma básico de MBD pode ser entendido como

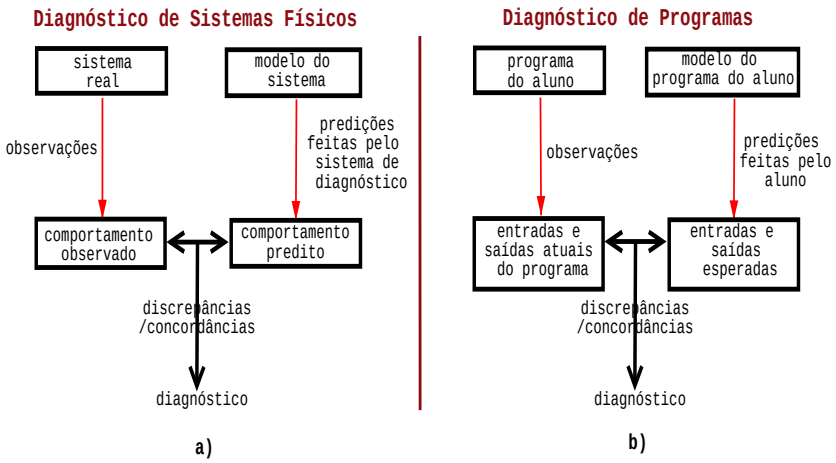


Figura 8: Diagnóstico baseado em modelos visto como a interação de observações e predições de Sistemas Físicos e de Programas

a interação de observações e predições (Figura 8(a)) [Benjamins, 1993]. Por um lado, temos o 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.

Um tipo de modelo que pode ser usado em MBD é o modelo baseado em componentes, o qual 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*).

De acordo com o tipo de inferência que o sistema de diagnóstico faz, este pode ser classificado em: *baseado em consistência* ou *abdutivo*. Em sistemas que aplicam o método abdutivo, a solução do diagnóstico tem a característica que todas as observações são consequência lógica do diagnóstico, i.e., a descrição do sistema junto com o diagnóstico implicam logicamente as observações. Por outro lado, em sistemas baseados em consistência, a solução do diagnóstico é consistente com as observações (a descrição do sistema junto com o diagnóstico e as observações são logicamente consistentes) [Benjamins, 1993].

Muitos pesquisadores têm desenvolvido sistemas para diagnóstico a partir da abordagem dos Princípios Básicos, usando uma representação da linguagem geralmente baseada em lógica de primeira ordem. Entre eles se destacam: a proposta de Reiter [Reiter, 1987] e a proposta de de Kleer [de Kleer and Williams, 1987] que usam o conceito de *conjuntos conflito* como base de seu método.

Em [de Kleer and Williams, 1987] é descrita uma *Máquina de Diagnóstico Geral*, chamada GDE, que pode detectar falhas múltiplas e simples. O processo de diagnóstico em GDE consiste de três fases: reconhecimento de conflitos, geração de candidatos e discriminação de candidatos.

GDE trabalha com conjuntos conflito minimais que são transformados em candidatos minimais. Reiter [Reiter, 1987] descreveu um método com o mesmo objetivo construindo uma árvore de conjuntos de corte.

7.1 Diagnóstico baseado em consistência

[Reiter, 1987] exemplifica a abordagem do diagnóstico baseado em consistência cuja formulação do problema e algoritmo apresentamos a seguir.

7.1.1 Formulação do problema

Definição 1: [Reiter, 1987]: O *sistema de diagnóstico* é um par (SD, COMP) onde:

- SD, a *descrição do sistema (o modelo)*, é um conjunto finito de sentenças de primeiro ordem.
- COMP, os *componentes do sistema*, é um conjunto finito de constantes.

Definição 2 [Reiter, 1987]: A *observação* de um sistema OBS é um conjunto finito de sentenças de primeiro ordem. A tripla (SD, COMP, OBS) é chamada de um *problema de diagnóstico* para o sistema (SD, COMP) com observações OBS.
A descrição do sistema faz uso dos predicados $AB(C)$ e $\neg AB(C)$ para especificar o comportamento anormal ou normal do componente $C \in COMP$

Definição 3 [Reiter, 1987]: 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 diagnóstico. O método de Reiter para calcular o diagnóstico faz uso dos conceitos de conjunto conflito e conjuntos de corte.

Definição 4 [Reiter, 1987]: Um *conjunto conflito* 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 [Reiter, 1987]: Seja C uma coleção de conjuntos. Um *conjunto de corte(hitting sets)* para C é um conjunto $H \subseteq \cup_{S \in C} S$ tal que $\forall S \in C, H \cap S \neq \emptyset$ (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 um conjunto de corte.

Teorema 1 [Reiter, 1987]: 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 conflito.

7.1.2 Algoritmo de Reiter

[Reiter, 1987] descreve o algoritmo que calcula os conjuntos de corte minimais para uma família de conjuntos F, algumas correções foram feitas por [Greiner et al., 1989].

O algoritmo gera um grafo acíclico com nós etiquetados por conjuntos e arcos etiquetados por elementos do conjunto. A idéia é que para cada nó etiquetado por um conjunto S , os arcos saindo dele são etiquetados por os elementos de S. Seja H(n) o conjunto formado pelas etiquetas do caminho da raiz até o nó n. O nó n tem que ser etiquetado por um conjunto S tal que $S \cap H(n) = \varnothing$. Se nenhum conjunto pode ser encontrado, o nó é etiquetado com @. A idéia é que qualquer caminho terminado com um nó etiquetado por @ é um conjunto de corte, dado que intercepta todas as possíveis etiquetas dos nós. O algoritmo tenta gerar a menor quantidade de novas etiquetas de nós como seja possível, dado que para diagnóstico, a família de conjuntos F, que é usada para etiquetar nós, não é explicitamente conhecida, calcular um elemento de F envolve uma chamada ao provador de teoremas para achar um conjunto conflito, é por tanto é uma operação muito cara. O algoritmo minimiza o número de chamadas ao provador de teoremas fazendo poda ao grafo enquanto está sendo construído. Quando um nó novo tem que ser etiquetado o algoritmo tenta primeiro reusar etiquetas existentes. Se uma etiqueta de nó S é um superconjunto de outra etiqueta S', então este pode ser fechado, dado que qualquer conjunto de corte para F será um conjunto de corte para $F \setminus S$ [Wassermann, 1999].

O algoritmo de Reiter modificado, que expande o grafo primeiro em largura, é mostrado a continuação:

1. Escolha um elemento de F para rotular a raiz (nível 0)
2. Para cada nó n do nível i fazer:
   (a) Se n tem rótulo S então para cada $s \in S$ crie um *s_arco* saindo de n com etiqueta s.
   (b) H(n)= conjunto de rótulos do caminho da raiz até n.
   (c) Se para algum n' temos $H(n') = H(n) \cup \{s\}$ 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 $S' \cap (H(n) \cup \{s\}) = \varnothing$
   o s_arco aponta para um novo nó com rótulo S'.
   (f) Senão faça o s_arco apontar para um novo nó m é rotule m com o primeiro elemento S' de F tal que $S' \cap H(m) = \varnothing$ (para diagnóstico isto é feito com ajuda do provador de teoremas). Senão existir, use um rótulo especial @
      i. se existe n' com rotulo S* tal que $S' \subset$ S* 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

Teorema: Os cortes minimais de F são {H(n) tal que n tem rótulo @}
Algumas considerações adicionais de implementação do algoritmo de Reiter devem ser tomadas em conta, dado que, como falamos anteriormente, a família de conjuntos F não é explicitamente conhecida e é calculada de a poucos, usando um provador de teoremas:

- Para gerar os rótulos vamos usar um provador de teoremas que devolva um conjunto *I* de elementos inconsistentes. O conjunto conflito é
  $C = I \cap COMP$.
- Se o provador de teoremas devolve o conjunto vazio então o nó é rotulado com @.
- O primeiro nó é rotulado com a intersecção entre COMP e a resposta devolvida por o provador de teoremas que tem como entrada $SD \cup COMP \cup OBS$
- Os demais nós são rotulados com a intersecção entre COMP e a resposta devolvida por o provador de teoremas que tem como entrada $SD \cup (COMP \setminus H(m)) \cup OBS$.

7.1.3 Exemplo de aplicação do Algoritmo de Reiter

Considere um sistema de alarme de carro que avise, tocando uma buzina, quando o carro circula com uma porta aberta ou quando está estacionado com as luzes acesas (o circuito de tal sistema é mostrado na figura 9).

Seja:

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

- a descrição do sistema (*System Description: SD*) é:

SD= {

$P \wedge C \wedge OKX \rightarrow I$ $\quad\quad$ $\neg(P \wedge C) \wedge OKX \rightarrow \neg I$

$C \wedge OKY \rightarrow \neg K$ $\quad\quad$ $\neg C \wedge OKY \rightarrow K$

$K \wedge L \wedge OKW \rightarrow J$ $\quad\quad$ $\neg(K \wedge L) \wedge OKW \rightarrow \neg J$

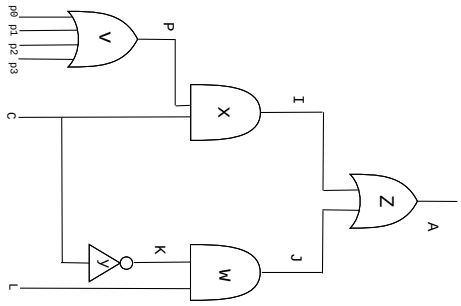


Figura 9: Circuito de um alarme de carro

$(I \vee J) \wedge OKZ \rightarrow A$ $\quad \neg(I \vee J) \wedge OKZ \rightarrow \neg A$

$(P0 \vee P1 \vee P2 \vee P3) \wedge OKV \rightarrow P$ $\quad \neg(P0 \vee P1 \vee P2 \vee P3) \wedge OKV \rightarrow \neg P$

}

- O conjunto de componentes COMP do sistema é composto pelas portas lógica OR, AND e NOT do circuito.

  COMP={ OKX, OKY, OKW, OKZ, OKV}

- com o alarme instalado, observamos que o carro está estacionado, as luzes não estão acesas, a porta P0 está aberta e o alarme está tocando. Essa descrição caracteriza uma observação (OBS) do estado atual do sistema.

  OBS={ $P0, \neg C, \neg L, A$}

A observação OBS reflete um comportamento anormal do sistema com relação ao projeto original do sistema de alarme, indicando a necessidade de se fazer o diagnóstico.

Aplicando o algoritmo de Reiter, descrito acima, temos como resposta o Diagnóstico

Δ={{ OKX}, {OKW}, {OKZ} }

Todo o processo para achar o diagnóstico é mostrado na Figura 10. Na figura, o primeiro nó (parte a) é rotulado com a resposta devolvida pelo provador de teoremas que tem como entrada $SD \cup COMP \cup OBS$, que neste exemplo é $\{OKX, , OKY, OKW, OKZ, OKV\}$. 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 $\{OKX, , OKY, OKW, OKZ, OKV\}$. 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 o conjunto vazio quando considera o componente OKX falho. Para o s_arco seguinte o provador de teoremas devolve $\{OKX, OKW, OKZ, OKV\}$ e de acordo com o passo 2.f.i, devemos mudar o rótulo do nó raiz para $\{OKX, OKW, OKZ, OKV\}$ e remover todos os arcos partindo de OKY (Figura 10 b)).

A Figura 10, parte c), mostra que para o s_arco *OKW* e *OKZ* são criados nós rotulados com @ e para o s_arco *OKV* é criado um nó rotulado por $\{OKX, OKY, OKW, OKZ\}$, de acordo com a resposta do provador de teoremas, supondo falho o componente *OKV*. Para o nível seguinte, criamos um arco *OKX* que será podado uma vez que gera uma suposição de falha já analisada anteriormente (de acordo com o passo 2.d do algoritmo). Para *OKY* criamos o nó $\{OKX, OKW, OKZ\}$ (passo 2.f.i). Finalmente na parte d) da figura, mudamos o rótulo da raiz para $\{OKX, OKW, OKZ\}$ e removemos todos os nós gerados a partir do s_arco *OKV* que foi eliminado da raiz.

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

Δ={{ OKX}, {OKW} , {OKZ} }

7.2 Diagnóstico do Programa do Aluno

A idéia básica é derivar um modelo diretamente do próprio programa e da semântica da linguagem de programação. Este modelo tem que distinguir componentes, descrever seus comportamentos e a estrutura do programa. Dado que o sistema a ser diagnosticado é um programa, a descrição do sistema é uma descrição do comportamento do programa do aluno (Figura 8(b)), que é derivado do programa e da semântica da linguagem em que o programa foi escrito.

É importante notar que em problemas de diagnóstico tradicionais de sistemas físicos o modelo está correto e a observação reflete o comportamento incorreto. Na proposta de depuração baseada em modelos, os papéis são invertidos: a descrição do sistema reflete o programa do aluno com erros e as observações (casos de teste) são saídas incorretas, i.e, as saídas observadas no programa do aluno. Já as predições a partir do modelo, não são feitas pelo sistema de diagnóstico mas sim pelo próprio aluno (Figura 11). É nessa situação que se espera que o aluno seja capaz de comunicar suas intenções de programação, que podem incluir conceitos errados sobre o uso da linguagem ou do funcionamento do computador.

Enquanto no modelo estrutural de sistemas físicos (Figura 9), os componentes são as portas lógicas e as conexões são as entradas e saídas das portas, no caso de programas, os componentes são as sentenças da linguagem (Figura 13).

No processo de diagnóstico proposto por Mateis *et al.* [Mateis et al.,], o programa é compilado em uma representação interna e junto com os fragmentos do modelo (descrição lógica das partes do modelo) eles são convertidos em modelos formais. A construção de modelos é feita automaticamente. O modelo junto com o comportamento especificado do programa (a partir de casos de teste) são usados pelo mecanismo de diagnóstico geral para encontrar os candidatos causadores do erro. Existem duas propostas de modelagem de programas: *modelo baseado em valor* [Cristinel Mateis and Wotawa, 2000] e *modelo baseado em dependências* [Wieland, 2001]. Ambas usam a abordagem hierárquica para diagnosticar programas feitos em Java e são usadas para detectar os erros do código fonte que se manifestam como

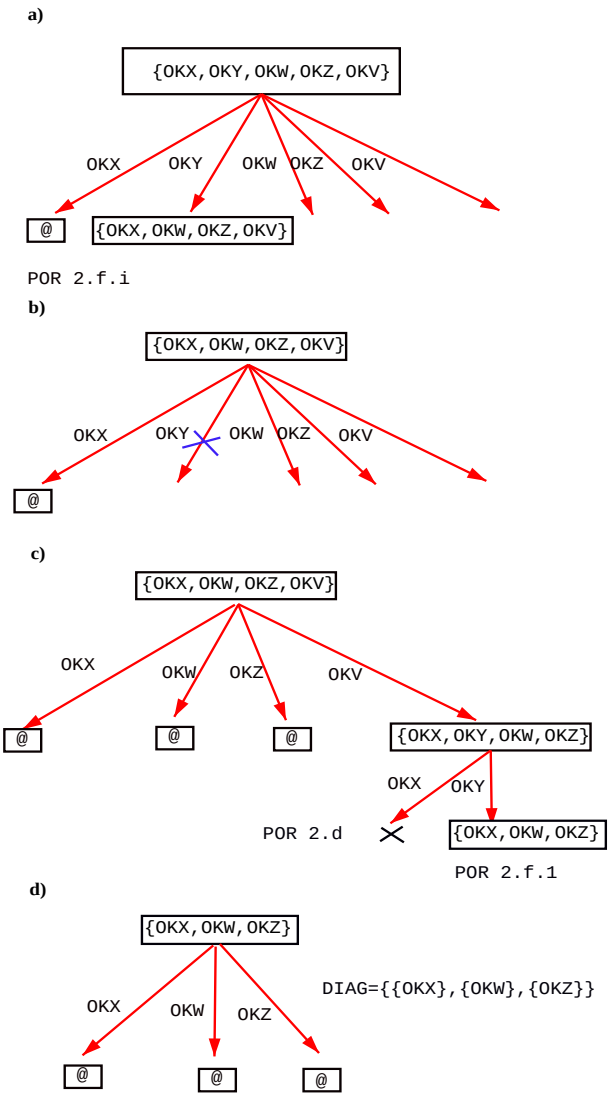


Figura 10: Aplicação do algoritmo de Reiter para o circuito de um alarme de carro

	problemas de diagnóstico tradicionais de sistemas físicos	**depuração de programas baseada em modelos**
modelo	correto	incorreto
observação	reflete o comportamento incorreto	reflete o comportamento correto do programa incorreto do aluno
predição	predições feitas pelo sistema de diagnóstico	predições feitas pelo aluno

Figura 11: Na depuração baseada em modelos os papéis são invertidos

um erro 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 [Wieland, 2001].

Falhas funcionais são falhas que surgem do armazenamento de um valor incorreto de alguma variável, pelo menos em uma execução possível da avaliação. Em particular, estas falhas incluem o uso de um operador incorreto ou uso de literais incorretos (variáveis). Exemplos de erros funcionais são:

- omitir um operador (e.g., escrever i em vez de i+1);
- usar um operador incorreto (e.g., escrever i++ em vez de ++i) ou a variável errada (e.g., a[i] em vez de a[j]);
- não inicialização de variáveis;
- uma modificação errada de um valor armazenado em uma variável;
- erros com inicializações das sentenças do laço; condições de saída que conduzem a um valor errôneo de uma variável.

Falhas estruturais são os erros do código fonte que alteram a estrutura do programa do aluno. Por exemplo: esquecer uma sentença, adicionar sentenças desnecessárias ou fazer uma atribuição a uma variável incorreta.

7.3 O Modelo Baseado em Valor

No modelo baseado em valor de um programa, expressões e sentenças são representadas como componentes, com a semântica das expressões e sentenças descritas por um conjunto de sentenças lógicas. Os componentes estão conectados se houver um fluxo de informação entre as expressões correspondentes e as sentenças (Figura 12).

Para obter o modelo estrutural Stumptner *et al.* [Cristinel Mateis and Wotawa, 2000] fazem as seguintes conversões:

- todas as variáveis são mapeadas a conexões e sempre que uma variável ocorre em uma expressão (componente), esta conexão é usada para conectar os componentes correspondentes. Cada vez que uma variável é usada no lado esquerdo de uma atribuição uma nova conexão e criada e usada para todos os componentes que usam essa variável até que a variável seja usada outra vez no lado esquerdo de uma atribuição;
- todas as sentenças: atribuições, condicionais, laços while, sentenças return, chamadas a métodos e expressões, são mapeadas em componentes.

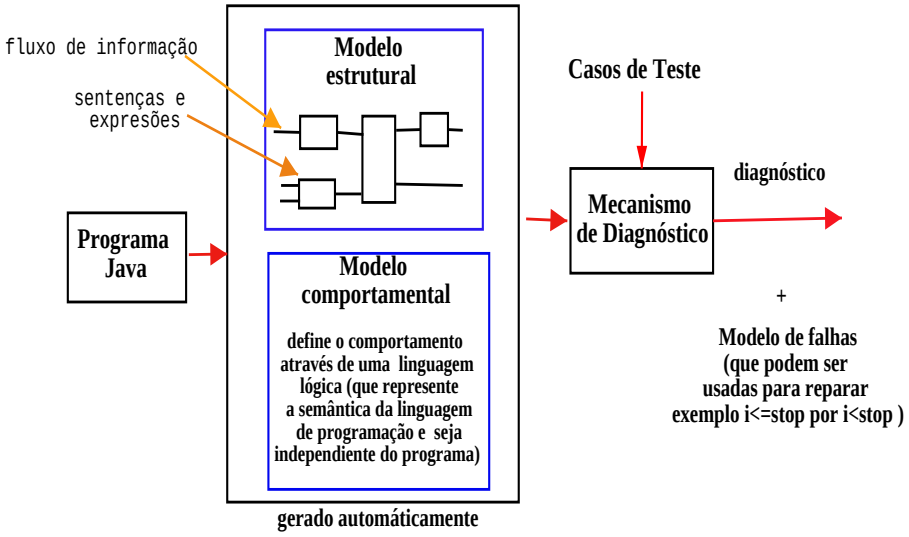


Figura 12: Processo do modelo baseado em valor

O fluxo de informação entre uma atribuição e uma outra sentença ocorre, por exemplo, se a atribuição muda a variável que é usada por outra sentença e não há nenhuma atribuição que mude a mesma variável entre elas. Um exemplo de modelo estrutural baseado em valor de um fragmento de programa é mostrado na Figura 13.

7.4 O Modelo baseado em dependências

A dependência funcional descreve o fato que o valor de uma determinada variável em um ponto particular dentro do programa depende de alguns outros valores de variáveis, constantes, ou métodos.

Os tipos de dependências indicadas pelo Dominik Wieland [Wieland, 2001] são: dependências de dados, dependências de controle e dependências potenciais.

- **Dependências de dados** entre a ocorrência de variável v1 e v2 ocorre se v1 está no lado esquerdo da atribuição da variável e v2 aparece no lado direito da atribuição. Então dizemos que o valor de v1 depende do valor de v2. Por exemplo: $v1 = 2 * v2$.

- **Dependência de controle** ocorre em uma sentença de laço ou seleção, onde o valor da avaliação da condição determina as instruções a serem executadas. Uma dependência de controle entre duas ocorrências de variáveis v1 e v2 existe se o valor de v1 é mudado em um ramo da sentença de seleção ou no corpo do laço e v2 aparece na condição do laço ou seleção.

- **Dependências potenciais** ignoram dependências em tempo de execução e focalizam-se em todas as influências de variáveis as quais possivelmente ocorreram no tempo de execução. Imagine uma

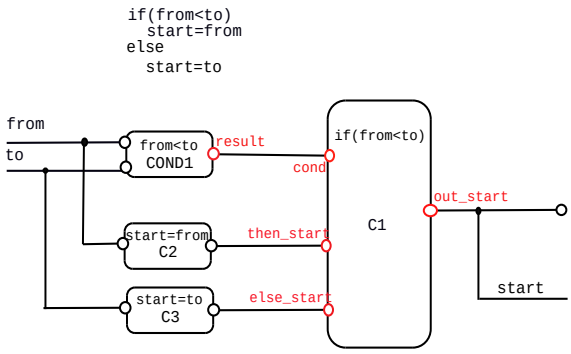


Figura 13: Exemplo do modelo estrutural baseado em valor de um fragmento de programa.

dependência de dados entre a ocorrência da variável v1 e v2 originada por uma atribuição de variável em um dos ramos de uma sentença de seleção. Se no tempo de execução este ramo não é executado, a influência de dado acima mencionada não será atingida. No entanto, do ponto de vista estático, esta constitui uma dependência potencial, que poderia afetar o valor de v1 em tempo de execução.

Três modelos de dependência funcionais concretos são introduzidos no [Wieland, 2001]:

- **Modelo de dependência funcional de avaliação de traço** (*Evaluation Trace functional dependency model ETFDM*) faz uso de informação em tempo de execução e pode conseqüentemente determinar todas as dependências de dados e de controle atingidas no tempo de execução.
- **Modelo de dependência funcional detalhado** (*Detailed functional dependency model DFDM*) é um modelo puramente estático que leva em conta todos os possíveis cenários do tempo de execução (influências potenciais).
- **Modelo de dependência funcional simplificado** (*Simplified functional dependency model SFDM*) é baseado no ETFDM ou no DFDM e pode ser interpretado como um modelo mais abstrato, possuindo as seguintes características: é mais fácil de ler e compreender devido a sua estrutura mais simples; é menos exato e menos detalhado do que os outros modelos.

O Modelo Baseado em Valor trabalha no nível de expressões e sentenças, permitindo uma localização de erros mais exata porém, a quantidade de conexões do sistema é maior. Já os Modelos Baseados em Dependência trabalham no nível de sentenças. Dado que o modelo baseado em valor pode eliminar diagnósticos errados usando informação adicional de tempo de execução (os valores das variáveis), este consegue melhores resultados do que o modelo baseado em dependências na maioria de

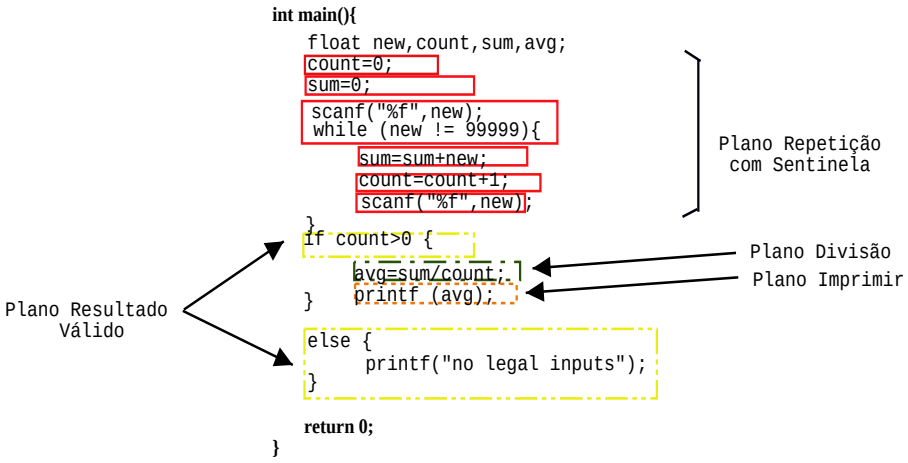


Figura 14: Planos usados para o problema 1

casos [Chen and Wotawa, 2003]. Por esta razão, o módulo de diagnóstico de ProPAT usará o enfoque do modelo baseado em valor para a depuração automática do programa do aluno.

7.5 Padrões Pedagógicos como Componentes no Modelo do Programa

Um padrão também pode ser modelado como um componente com entradas e saídas (Figura 15). Desta forma, o processo de depuração, além de modelar o programa como foi descrito na seção 7.3, inclui no modelo os padrões para identificar as intenções do estudante. Note que, nossa proposta é que o tutor ProPAT faça um diagnóstico hierárquico, isto é, o sistema de diagnóstico deve trabalhar em níveis hierárquicos de detalhamento dos componentes do programa do aluno, e assim ser capaz de detectar falhas no nível de padrões (estratégias ou planos de alto nível) ou no nível de sentenças (Figura 15). Podemos resolver o Problema 1 usando os seguintes padrões: *Repetição com Sentinela*, *Divisão*, *Imprimir*, *Resultado Válido* como é mostrado na Figura 14.

Na Figura 15 mostramos o modelo de componentes para o Problema 1, identificando o padrão *Repetição com Sentinela* como um componente. Note que podemos construir dois modelos distintos do mesmo programa: um contendo somente os componentes relacionados às sentenças da linguagem e outro contendo componentes relacionados aos padrões identificados no programa do aluno.

8 Conclusões

Neste trabalho é proposta a construção do módulo de diagnóstico para um Sistema Tutor Inteligente para Programação, denominado ProPAT, que combina duas técnicas de Inteligência Artificial: *Diagnóstico Baseado em Modelos* (MBD) e *Diagnóstico Baseado em Intenções* para detectar erros no programa

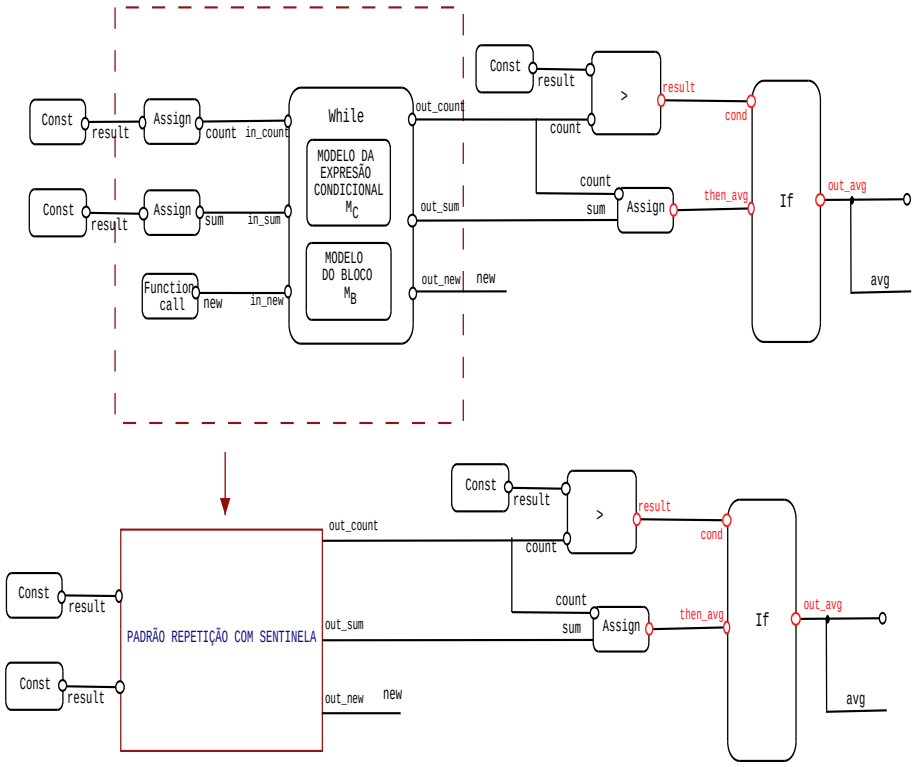


Figura 15: Modelo de componentes e conexões para o programa 1

do aluno. O diagnóstico de programas baseado em modelos analisa um modelo do programa do aluno representado na forma de componentes e conexões, onde os componentes correspondem às estruturas lógicas da linguagem. A abordagem baseada em intenções, usada para reconhecer o plano ou estratégia de solução do aluno, é implementada através do uso de padrões pedagógicos - padrões de programação recomendados para aprendizes. Uma das contribuições dessa proposta de trabalho é o uso de padrões como componentes para o diagnóstico baseado em modelos. Combinando estas duas técnicas acreditamos que ProPAT herda as vantagens de ambas: a capacidade de detectar falhas conceituais, refletidas num erro do projeto do programa, e falhas funcionais no nível de sentenças. É feita uma suposição importante nesta proposta: que o professor apresentará o conjunto de Padrões Pedagógicos aos estudantes em sala de aula. Assim eles estarão motivados a aplicar os mesmos padrões para resolver problemas novos, restando ao sistema de diagnóstico a tarefa de detectar falhas de implementação do padrão.

Referências

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

[Astrachan and Wallingford, 1998] Astrachan, O. and Wallingford, E. (1998). Loop patterns. `http://www.cs.duke.edu/~ola/patterns/plopd/loops.html`.

[Benjamins, 1993] Benjamins, R. (1993). *Problem Solving Methods for Diagnosis*. PhD thesis, PhD thesis, University of Amsterdam.

[Bergin, 1999] Bergin, J. (1999). Patterns for selection version 4. `http://csis.pace.edu/~bergin/patterns/Patternsv4.html`.

[Bridgeman, 2002] Bridgeman, S. (2002). Intro to computing i. `http://cs.colgate.edu/faculty/stina/courses/cosc/101/f02/syllabus.html`.

[Chen and Wotawa, 2003] Chen, R. and Wotawa, F. (2003). Debugging with an enriched dependency-based model or how to distinguish between aliasing and value assignment. In *Seventeenth International Workshop on Qualitative Reasoning*.

[Cristinel Mateis and Wotawa, 2000] Cristinel Mateis, M. S. and Wotawa, F. (2000). A value-based diagnosis model for java programs. In *Eleventh International Workshop on Principles of Diagnosis (DX)*. `http://www.dbai.tuwien.ac.at/staff/wotawa/dx2000c.ps.gz`.

[de Kleer and Williams, 1976] de Kleer, J. and Williams, B. C. (1976). Local methods of localizing faults in electronic circuits. *MIT.AI Lab Memo*, 394.

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

[Dingle and Zander, 2001] Dingle, A. and Zander, C. (2001). Assessing the ripple effect of cs1 language choice. *J. Comput. Small Coll.*, 16(2):85–93.

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

[East et al., 1996] East, J. P., Thomas, R., Wallingford, E., Beck, W., and Drake, J. (June 1996). Pattern-based programming instruction. In *in the Proceedings of the ASEE Annual Conference and Exposition, Washington, DC*.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Adisson Wesley.

[Greiner et al., 1989] Greiner, R., Smith, B. A., and Wilkerson, R. W. (1989). A correction to the algorithm in reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88.

[Johnson and Soloway, 1984] Johnson, W. L. and Soloway, E. (1984). Proust: Knowledge-based program understanding. In *In Proceedings of the 7th international conference on Software engineering, Florida, United States*, pages 369 – 380.

[Kumar, 2002] Kumar, A. (2002). Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug c++. In *In Proceedings of Intelligent Tutoring Systems, LNCS 2363,France*, pages 792–801.

[Mateis et al.,] Mateis, C., Stumptner, M., Wieland, D., and Wotawa, F. Java ai support for debugging java programs. `http://citeseer.ist.psu.edu/383111.html`.

[Mozetic, 1992] Mozetic, I. (1992). Model-based diagnosis: An overview. In *Advanced Topics in Artificial Intelligence*, pages 419–430.

[Porter and Calder, 2003] Porter, R. and Calder, P. (2003). A pattern-based problem-solving process for novice programmers. In *Proceedings of the fifth Australasian conference on Computing education*, pages 231–238. Australian Computer Society, Inc.

[Proulx, 2000] Proulx, V. K. (2000). 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.

[Reiter, 1987] Reiter, R. (1987). A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95.

[Self, 1993] Self, J. (1993). Model-based cognitive diagnosis. In *AAI, AI-ED Technical Report No 82*. http://citeseer.nj.nec.com/self93modelbased.html.

[Stumptner and Wotawa, 1998] Stumptner, M. and Wotawa, F. (1998). A survey of intelligent debugging. *AI Communications 11*, pages 35–51. http://citeseer.nj.nec.com/stumptner98survey.html.

[Wallingford, 2001] Wallingford, E. (2001). The elementary patterns home page. http://www.cs.uni.edu/~wallingf/patterns/elementary/.

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

[Wenger, 1987] Wenger, E. (1987). *Artificial intelligence and tutoring systems: computational and cognitive approaches to the communication of knowledge*. Morgan Kaufmann Publishers Inc.

[Wieland, 2001] Wieland, D. (2001). *Model-Based Debugging of Java Programs Using Dependencies*. PhD thesis, PhD Thesis,Technische Universität Wien.