Autoria e Interpretação Tutorial de Soluções Alternativas para Promover o Ensino de Programação de Computadores

Gabriel dos Santos, Alexandre Ibrahim Direne, André Luiz Pires Guedes

Departamento de Informática, Universidade Federal do Paraná, Brasil
CEP 81531-990 - Curitiba - PR - Brasil
{gabriel,alexd,andre}@inf.ufpr.br

Resumo

O presente artigo aborda linguagens e ferramentas de autoria assim como Sistemas Tutores Inteligentes (STI) para apoiar o ensino de programação de computadores. Até então, nenhum trabalho apresentou um sistema de autoria para desenvolvimento de STIs nessa categoria e nem ferramentas indenpentes da linguagem ensinada. Além disso, as ferramentas existentes ou restringem demais a criatividade do aluno ou falham em prover um feedback com alto valor cognitivo agregado. Esse trabalho aborda a construção de ambas as ferramentas, procurando apresentar abordagens que tratem de ambas as limitações mencionadas.

Palavras-Chave: Ambientes Interativos de Aprendizagem, Modelagem Cognitiva Aplicada à Educação, Inteligência Artificial Aplicada à Educação, Linguagens e Ferramentas de Autoria.

Abstract

This paper approaches languages and authoring tools as well as Intelligent Tutoring Systems (ITS) to support teaching of computer programming. So far, no work presented an authoring system to develop tutors for programming and neither have anyone developed a language-independent tool. Besides, the existing tools either restrict too much the student's creativity or fail to provide a feedback useful for cognitive development. This work presents approaches to address both limitations.

Key Words: Interactive Environments for Learning, Cognitive Modeling Applied to Education, Artificial Intelligence Applied to Education, Languages and Tools for Authoring.

1 Introdução

Nesse trabalho serão apresentadas a arquitetura de um sistema de autoria e a de um interpretador tutorial ambos para o ensino de linguagens de programação. Dentro desse contexto, estarão enfocados alguns dos elementos já implementados, dando particular ênfase para uma linguagem de autoria de padrões de erro ou acerto e de como esses padrões podem ser utilizados no contexto geral como uma ferramenta bastante poderosa e flexível, enriquecendo o valor cognitivo que pode ser agregado aos cursos desenvolvidos pelo sistema.

Ensinar, de uma forma geral, é uma tarefa de enorme complexidade, e os processos cognitivos envolvidos ainda são desconhecidos mesmo para profissionais humanos. Apesar de existirem metodologias pedagógicas, poucas são aceitas universalmente, e mesmo as mais consolidadas não formalizam um processo claro de procedimento de ensino, deixando muitos aspectos delegados ao instrutor. A tarefa de programação, em particular, demanda uma enorme carga cognitiva, uma vez que não se restringe apenas à capacidade de trabalhar com uma linguagem de programação[du Boulay and Sothcott 1988].

Podemos dividir a atividade sob dois aspectos: a aquisição de conhecimentos de princípios de programação e a perícia em programação[du Boulay and Sothcott 1987]. O conhecimento de

princípio abrange os detalhes dos rígidos formalismos sintático e semântico das estruturas linguísticas de uma Linguagem de Programação (LP), ou seja, a compreensão da forma de escrita e dos resultados. O conhecimento de **perícia** abrange a habilidade necessária para integrar detalhes sintáticos e semânticos isolados em um planejamento maior que agregue vários fragmentos para compor um programa completo. Essa habilidade às vezes é também conhecida como habilidade tática[du Boulay and Sothcott 1987], e considera justamente as habilidades de decomposição de problemas e de composição de soluções a partir de fragmentos que correspondem a soluções de problemas menores.

Sistemas tutores inteligentes e ambientes exploratórios para aprendizagem de programação de computadores procuraram abordar estes dois tipos de conhecimento (princípio e perícia) por meio de diversas técnicas distintas. Dentro do assunto abordado por este trabalho, alguns trabalhos contemplaram o diagnóstico de programa do aluno [Adam and Laurent 1980, Goldstein 1975, Hasemer 1983, Johnson 1986] e foram contribuições importantes para a área. Entretanto, nenhuma das contribuições de pesquisa de representação de soluções de problemas se preocupou em adotar abordagens genéricas de representação de soluções de problemas por meio de linguagens e ferramentas de autoria. Dadas as limitações dos métodos e ferramentas para o diagnóstico de programas de aprendizes em particular, torna-se importante a possibilidade de dar espaço amplo à atuação humana, por meio de autoria, para apresentar variações de programas que solucionam um mesmo problema. Ainda, é importante que o engenho de avaliação seja genérico através de uma representação independente de linguagem, abrindo espaço para criação de cursos em linguagens diversas para o mesmo engenho. Poucos sistemas, tais como o sistema do Prottel[Direne 1997] e o Demonstr8[Blessing 1997], tentaram construir mecanismos genéricos e nenhum deles abordou a dimensão de ferramentas associadas ao ensino de programação de computadores, ambos objetivos desse trabalho.

2 Trabalhos Correlacionados

2.1 Sistemas Tutores Inteligentes

O conceito de Sistema Tutor Inteligente surgiu da idéia de aplicar técnicas de Inteligência Artificial (IA) em programas com objetivos educacionais. Dessa forma, STIs são programas de computador projetados para incorporar técnicas provenientes do campo de IA de modo a prover tutores que saibam *como* ensinar, *o que* ensinar e conheçam a *quem* eles estão ensinando.

2.2 Linguagens e ferramentas de autoria

Uma vez que um dos objetivos do trabalho proposto é o de desenvolver uma ferramenta de autoria para tutores de programação, não poderíamos deixar de mencionar outros trabalhos de autoria de STI existentes.

Um sistema de autoria de STI tem como objetivo prover uma ferramenta relativamente genérica para construção de sistemas tutores minimizando a necessidade de conhecimento fora do domínio a ser tutorado, dessa forma tornando a ferramenta acessível para especialistas do domínio. Uma das principais resenhas neste campo, e à qual se deve muito no sentido de divulgar o conhecimento do estado da arte em autoria de STI, é a desenvolvida por Murray[Murray 1999], na qual é apresentada uma categorização de vários dos sistemas existentes. Embora o trabalho de Murray não contemple todos os sistemas de autoria já construídos, um aspecto importante é que este provê uma amostragem que abrange a área como um todo. Murray divide os sistemas de autoria segundo o tipo de tutores que produz. As categorias são as seguintes: Planejamento e sequenciamento de currículo; Estratégias tutoriais; Simulação de dispositivos e treinamento com equipamentos; Sistema especialista no domínio; Sistemas de propósito específico.

Uma vez que não é objetivo deste trabalho, não vamos detalhar cada categoria, a não ser a de *Sistemas Especialistas no Domínio*. Um Sistema Especialista no Domínio é, a grosso modo, o tipo de sistema que tem conhecimento e de alguma forma é capaz de interpretar o domínio a ser tutorado. Dois trabalhos interessantes se destacam nessa categoria, o **Demonstrat8** de Blessing[Blessing 1997] e o **RIDES** de Munro[Munro et al. 1997]. A característica comum de ambos os sistemas é a forma de autoria através de demonstração.

Em um sistema de autoria por demonstração, o autor pode criar conteúdos pedagógicos e do domínio específico, apenas demonstrando como um determinado problema é solucionado. A autoria por demonstração, dentro do contexto de programação, pode ser um programa que o autor provê como exemplo de solução para um determinado problema.

O trabalho de Murray[Murray 1999], além de sua importante contribuição para resumir o estado da arte em sistema de autoria de STI, reforça a constatação do presente artigo de que não existe nenhum sistema de autoria para tutores de programação, portanto assinalando um dos pontos originais do trabalho que estamos desenvolvendo.

2.3 Diagnóstico Automático de Programas

Também conhecidos como diagnosticadores (*Bug Finders*), os sistemas de diagnóstico automático são ferramentas que têm como objetivo encontrar e classificar erros no programa provido pelo aprendiz. Isso é feito com o objetivo de orientar o aprendiz quanto aos erros cometidos, além de ser uma ferramenta muito importante para o próprio STI no preenchimento do modelo do aprendiz.

Essa categoria de trabalhos é de grande interesse científico porque cobre vários dos mais importantes projetos de pesquisa nos quais os programas, na medida do possível, são capazes de detectar discrepâncias no programa do aprendiz, tarefa que será tratada em parte pelo presente trabalho. Os sistemas de diagnóstico podem ser encaixados em três categorias[du Boulay and Sothcott 1987]:

- **Solução de Referência (Specimen Answer)** Sistemas que utilizam uma solução de referência para confrontar com a solução do aprendiz. Um exemplo de trabalho nesta área é o sistema LAURA[Adam and Laurent 1980].
- **Análise de Especificação (Specification Analysys)** Sistemas que avaliam o programa do aluno segundo a espeficicação do problema. Alguns exemplos de sistemas que utilizam análise de especificação são MYCROFT[Goldstein 1975], AURAC[Hasemer 1983], PUDSY[Lukey 1980] e PROUST[Johnson 1986].
- **Diálogo de Depuração (Debugging Dialogue)** Sistemas dessa categoria entram em um diálogo com o aluno, tentando orientá-lo a encontrar o erro. Um exemplo de sistema que cai nessa classe é o desenvolvido por Shapiro[Shapiro 1983], para trabalhar com linguagens declarativas (em particular, com o Prolog).

2.3.1 Solução de Referência

LAURA foi um sistema único no seu ramo. Tal como sugere sua categorização, LAURA se utiliza de uma solução de referência (um programa) para diagnosticar o programa do aluno. Para tanto, o sistema construía o grafo de fluxo de controle de ambos os programas. O estágio que se segue é uma tentativa de transformar o grafo do aluno e casar com o grafo de referência. São construídas várias hipóteses sobre a correlação entre os nodos e arcos do grafo. Uma vez que seja encontrada uma correspondência mais aceitável, tomam lugar uma série de transformações, tais como permutar passos independentes da ordem de computação.

Uma vez que o sistema se utiliza de um processo sofisticado de casamento de padrões, a grande vantagem de LAURA reside na capacidade de aceitar programas corretos, mesmo que não sejam idênticos ao programa de referência. Essa capacidade é importante, uma vez que permite que o aluno exerça em parte sua criatividade e não seja coagido a seguir um caminho específico. Por outro lado, o grande defeito reside na capacidade de resposta do sistema. Para o caso de existência de erros, LAURA é capaz apenas de diagnosticar erros em baixo nível de abstração, ou seja, erros pequenos que não levam em conta o contexto principal de solução do problema abordado no exercício em questão. Porque o sistema trabalha de forma genérica, as mensagens de erro são construídas dinamicamente, e portanto o sistema é incapaz de considerar o contexto e apresentar informações relativas à capacidade de planejamento do aluno. Esse defeito é inerente à abordagem específica do LAURA, e não aos mecanismos que analisam soluções de referência. Neste trabalho, adotamos uma abordagem capaz de amenizar ou até eliminar esse problema.

De qualquer forma, LAURA tem um papel fundamental dentro desse trabalho, uma vez que é o sistema que mais se assemelha à ferramenta de diagnóstico que nos propomos a desenvolver. Entraremos em mais detalhes de nossa solução na seção 3.

2.3.2 Diagnóstico por análise de especificação

Na classe de sistemas de Diagnóstico por Análise de Especificação (DAE), existe uma maior variedade de sistemas. O aspecto mais atraente da maioria dos sistemas dessa categoria é a capacidade do sistema de avaliar o programa do aluno de uma forma mais global, o que permite ao sistema prover explicações mais relevantes de acordo com o contexto do problema. Porque o espaço é limitado, estaremos discutindo apenas o sistema **PROUST**, embora todos os trabalhos tenham contribuído de forma tão significativa quanto o PROUST na categoria de diagnosticadores.

PROUST foi um sistema desenvolvido para diagnóstico de programas em Pascal, desenvolvido por Johnson *et al*[Johnson 1986], com resultados bastante impressionantes, tal como é tradicional dos sistemas dessa categoria. Nesse sistema, a especificação do programa é expressa através de um conjunto de objetivos. Cada objetivo é associado a um objeto que armazena informação declarativa sobre o mesmo, bem como relaciona um ou mais planos que atingem o objetivo proposto. Cada plano é associado a um template que pode ser usado para casamento com códigos em pascal, além da possibilidade de ter um conjunto de objetivos próprios, ou seja, apresentando assim uma definição recursiva.

A grande vantagem dessa abordagem é a capacidade de relacionar fragmentos de código aos objetivos presentes na especificação. Mais do que em qualquer outro sistema, o PROUST é capaz de fazer comentários impressionantes sobre problemas de lógica no programa, inclusive sendo capaz de sugerir dados de entrada para os quais o programa analisado não seria bem sucedido.

Por outro lado, em contraste com o LAURA, o PROUST é muito sensível a pequenas diferenças e incapaz de aceitar programas equivalentes, dessa forma restringindo muito os programas do aluno. Ainda, o PROUST é capaz de lidar apenas com um subconjunto do Pascal e, dentro desse contexto, apenas com um subconjunto de problemas.

2.4 Tipos de intervenção

Um sistema tutor de programação pode ser categorizado segundo os momentos nos quais é realizada uma intervenção. Veremos nessa seção um pouco de cada uma das duas abordagens utilizadas nos trabalhos publicados até então.

2.4.1 Intervenção durante o evento de programação

Nessa abordagem, o sistema monitora o aprendiz durante o processo de criação do programa. No momento em que o sistema detecta que o aluno está se desviando do caminho, imediatamente o sistema realiza uma intervenção. Entre os sistemas mais reconhecidos dentro desta categoria figuram **Malt** de Koffman *et al*[Koffman and Blount 1975], com ensino de linguagem de máquina, e o **Greaterp**, o tutor de LISP, desenvolvido por Anderson e seus colegas[Anderson et al. 1984, Reiser et al. 1985].

Esse tipo de sistema restringe bastante o aprendiz e impede que este tenha chance de experimentar ou até mesmo de oferecer uma solução correta alternativa ao problema. Para Anderson, a estratégia cognitiva por trás dessa abordagem é que o aluno não deve ter muita liberdade para seguir um caminho incorreto, com o receio de que aprenda o erro.

2.4.2 Intervenção posterior ao evento de programação

Nesse tipo de abordagem, o aluno fica livre para programar da forma como achar mais conveniente. O sistema só intervêm e comenta sobre o programa no momento em que o aluno assinalar que o solução está concluída. Todos os sistemas descritos na Seção 2.3 podem ser classificados sob essa categoria.

A atuação de sistemas nessa categoria tem por trás uma teoria cognitiva diferenciada, na qual é permitido ao aluno prosseguir e errar e somente interferir em um momento futuro. Se esta é ou não a melhor abordagem, é motivo de debate.

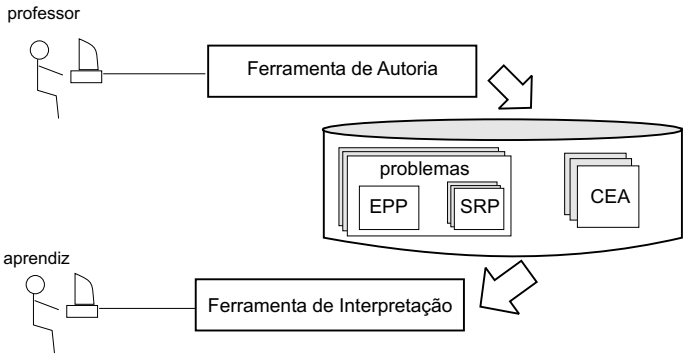

Figura 1: Arquitetura geral do Sistema

3 Arquitetura de uma Solução

Até então, discorremos sobre a complexidade do problema e fizemos uma breve revisão crítica de trabalhos que abordaram problemas semelhantes ou relacionados. O objetivo desta seção é justamente apresentar *como* os problemas mencionados serão abordados e qual a estrutura das ferramentas propostas na introdução.

Para que a apresentação da pesquisa seja mais frutífera, vale a pena abordar a arquitetura geral dos protótipos que serão construídos, tal como ilustrada na Figura 1. Nessa figura é possível observar as duas ferramentas propostas (uma de autoria e outra de interpretação tutorial), os atores que interagem com cada uma dessas ferramentas respectivamente (professor/autor e aprendiz) e os componentes criados e ou editados pela ferramenta de autoria, os quais são armazenados e utilizados como entrada na ferramenta de interpretação. Figuram entre estes componentes um conjunto de Classes de Erros do Aluno (CEA) e um conjunto de problemas propostos de programação, por sua vez constituídos do respectivo Enunciado de Problema Proposto (EPP) e de um conjunto de alternativas de soluções de referência para este problema (SRP). Dessa forma, o fluxo geral do sistema consiste na atuação do autor para, através da ferramenta de autoria, criar e armazenar material de curso, enquanto o aprendiz ativa a ferramenta de intrepretação e tenta resolver os problemas propostos no material.

3.1 Um Compilador-Mapeador

Para que o interpretador tutorial proposto seja capaz de analisar o programa provido pelo aluno, tanto as soluções de referência como a solução apresentada pelo aluno, são ambas representadas através de um Grafo de Fluxo de Controle (GFC), semelhante ao apresentado por Aho *et al*[Aho et al. 1986], a não ser por algumas peculiaridades, sendo a mais relevante o fato de que expressões aritméticas são representadas pelas suas respectivas árvores de derivação. Essa alteração se deve a maior flexibilidade para trabalhar e aplicar transformações algébricas a árvores de derivações do que a uma expressão representada por apenas um nodo.

A ferramenta para tradução de um programa em seu respectivo GFC já foi consolidada e presentemente está funcional. Essa ferramenta recebe como entrada um programa escrito em um dialeto da linguagem "C"e produz como resultado um GFC(no qual o grafo é descrito em *dot plain*[1]). O resultado do processamento é ilustrado na Figura 2.

3.2 Autoria

A ferramenta de autoria deve ser utilizada para criar e estender o conjunto de problemas que pode ser proposto ao aprendiz. A ferramenta de autoria proposta é relativamente simples, mas cumprirá o mínimo necessário para trabalhar como contrapartida do interpretador. A priori, para

[1]O *dotty*[Gansner and North 2000] é um software escrito para diversas plataformas usado para representar e renderizar grafos.

```
main()
{
  int b;

  for(f=0; 10−f; f=f+1){
    write(f);
  }

  write(f);
}
```

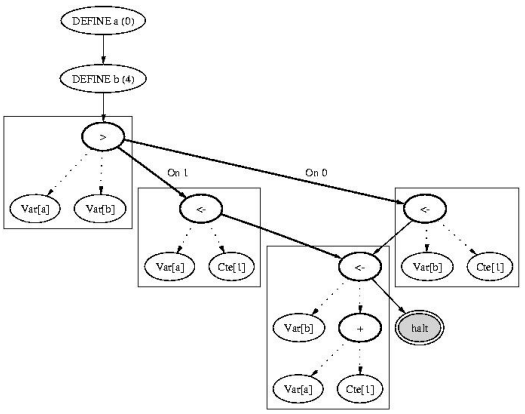

Figura 2: Exemplo de grafo de fluxo de dados representando um programa

fins de simplificação a ferramenta está restrita a trabalhar com programação em um subconjunto da linguagem "C", embora a ferramenta de interpretação seja independente de linguagem.

As principais primitivas da ferramenta de autoria são a criação de problemas e das respectivas soluções aceitáveis para o mesmo e a criação de operadores de descrição de erros através de um linguagem denominada nesse trabalho como LAPIDAE (Linguagem de Autoria de Padrões Interpretáveis Direcionados por Acerto ou Erro).

3.3 Interpretador Tutorial

Enquanto a autoria *produz o material* a ser utilizado pelo curso, o interpretador tutorial é responsável por manipular esse material, o qual pode incorporar de forma parcial ou integral o domínio de ensino, as estratégias, entre outros aspectos relativos ao curso. Ou seja, podemos dizer que o interpretador e o material constituiriam um Sistema Tutor Inteligente por completo. No trabalho proposto, a arquitetura concebida e o protótipo de interpretador a ser construído têm como objetivo apenas prover mecanismos de diagnóstico do programa do aluno. Uma vez que o interpretador realiza a análise através de GFCs, o sistema é indepentente de linguagem (mas não de paradigma, nesse caso restrito ao paradigma imperativo de programação).

3.4 LAPIDAE – Linguagem de Autoria de Padrões Interpretáveis Direcionados por Acerto ou Erro

Um padrão interpretável direcionado é definido nesse trabalho como sendo um padrão que representa um erro ou um acerto em um programa de entrada provido pelo aluno. Cada padrão pode ser composto, na verdade, de três partes, as quais serão apresentadas com detalhes no decorrer dessa seção. A LAPIDAE é utilizada em ambas as ferramentas mencinadas, como linguagem de descrição na ferramenta de autoria, enquanto o algoritmo que a processa faz parte da ferramenta de interpretação tutorial.

Sendo parte da ferramenta de autoria, uma questão a ser considerada é o quanto ela facilita o trabalho de construção de material de curso. No contexto de autoria de cursos de programação, o público alvo são professores e especialistas da área de programação. Dessa forma, uma ferramenta de autoria para programação não necessariamente é falha ao exigir que o usuário seja capaz de lidar com uma linguagem de padrões, uma vez que esta tem afinidade com linguagens de programação e de fato é muito menos complexa que tais linguagens.

3.4.1 Especificações da Linguagem

A linguagem criada para especificação de padrões é muito parecida com aquela utilizada pelo POP-11[Barrett et al. 1985] com a mesma finalidade, a não ser pela forma de especificação das entradas, que ao invés de listas de strings utiliza uma string apenas, tanto para representar o padrão como

para representar o texto a ser testado para casamento. Um padrão é uma sequência de meta-símbolos os quais devem ser utilizados para determinar que tipo de texto *satistaz* o padrão, ou ainda, as *restrições* que o padrão impõe para que algum texto de entrada seja aceito.

Os meta-símbolos que compõe a linguagem para especificação do padrão podem ser classificados de duas formas, baseado no tipo de casamento ou na capacidade de instanciar uma variável com o valor do casamento. Quanto ao tipo de casamento, o meta-símbolo pode especificar casamento exato, com uma string qualquer ou com uma sequência de zero a qualquer número de strings. Já do ponto de vista de instanciação, um meta-símbolo pode especificar se o segmento de texto casado deve ser atribuído a uma variável para uso futuro. Os meta-símbolos são descritos a seguir:

- Meta-símbolo literal ou constante: é uma string texto que deve casar com um valor específico no texto. Pode-se dizer que esses elementos é que delineiam a forma geral de um padrão. A representação desse meta-símbolo no padrão é qualquer string que não comece coms os caracteres '?' e '='.

- meta-símbolo de casamento unitário: indica que uma string qualquer pode ser casada com este símbolo. Para especificar que o valor casado deve ser atribuído a alguma variável, representa-se como $?NomeVarialARecebeValor$. Caso contrário, utiliza-se o símbolo =.

- meta-símbolo de casamento variável: indica que qualquer sequência de strings, inclusive uma sequência vazia, pode ser casada com este símbolo. Para especificar que o valor casado deve ser atribuído a alguma variável, representa-se como $??NomeVarialARecebeValor$. Caso contrário, utiliza-se o símbolo ==.

Tendo em mãos a descrição dos meta-símbolos, um exemplo de padrão é o seguinte:

```
== O rato ?acao a roupa do = de ??localizacao.
```

O padrão acima determina que os textos aceitos (casados) podem ter qualquer quantidade de strings seguidas de "O rato", depois por qualquer string, e assim por diante. Alguns exemplos de entradas e se casam ou não com o padrão apresentado:

- O rato roeu a roupa do rei de Roma na Itália. *[aceita]*

- Dizem que O rato roeu a roupa do rei de Roma. *[aceita]*

- O rato mastigou avidamente a roupa do rei de Roma. *[recusa]*

Um meta-símbolo vinculado a uma variável pode ser repetido para representar padrões mais complexos. Essa repetição tem como objetivo forçar a correspondência por meio da presença de segmentos idênticos no texto de entrada.

Completando os aspectos incorporados no padrão, embora ainda não completamente desenvolvido, a arquitetura foi concebida de forma a permitir que sejam adicionadas restrições a qualquer meta-símbolo (exceto o literal), restringindo as palavras (ou sequência de) que podem ser casadas com o mesmo. Para adicionar uma restrição, basta que, após um meta-símbolo, esteja presente o caracter ':' seguido do nome da funçao de restrição. Por exemplo, seria possível especificar que um determinado meta-símbolo deve casar com uma expressão regular ou com algum elemento sintático de uma linguagem de programação (como uma expressão aritmética).

Através dessa linguagem de padrão, um padrão de erro/acerto é definido por três partes (vide exemplo de padrão na Figura 3):

- Padrão de Entrada: esse padrão deverá ser casado no texto que representa um código fonte de um programa. As variáveis associadas a meta-símbolos do padrão de entrada podem ser usadas nos dois padrões descritos a seguir como forma de indicar que o texto casado deve ser substituídos no padrão em questão;

- Padrão de Saída: esse padrão representa um modelo de texto que deve ser usado para substituir o segmento de texto que for casado com o padrão de entrada. No caso do nosso sistema, uma substituição no segmento de texto pode representar uma substituição de um subgrafo do GFC que representa o programa;

Padrão entrada	Padrão saída	Explicação
if (??condicao:isExpression()) ?corpoIf:isStatementBlock()	**while** (??condicao) ?corpoIf	*Ao invés da utilização de uma iteração, você utilizou um condicional.*

Figura 3: Exemplos de padrões de erro.

- Padrão de Explicação: esse padrão representa um modelo de texto que será apresentado caso o sistema confirme a presença do padrão de entrada no programa fonte. Isso poderia indicar, por exemplo, que tipo de erro foi detectado.

3.4.2 Detalhes de Processamento

O problema de casamento de padrão é abordado como um problema de satistação de restrição, onde o padrão como um todo é interpretado como um conjunto de restrições que devem ser satisfeitas para que o texto de entrada seja casado com o padrão. Sendo um problema de satisfação de restrições, antes de resolvê-lo, é necessário identificar os conceitos básicos e modelar o problema de casamento como tal. A modelagem recai em responder as seguintes perguntas:

1. Quais são as variáveis do problema? Qual a natureza do valor dessas variáveis?

   Dado um meta-símbolo, este pode casar com um ou mais segmentos de texto distintos. A questão é com qual segmento de texto o meta-símbolo vai casar. Essa questão se traduz nas variáveis do nosso problema, e o objetivo final é justamente determinar o valor dessas variáveis tal que as restrições sejam satisfeitas.

2. Qual o domínio das variáveis?

   O domínio de uma variável que representa o casamento de um meta-símbolo é o conjunto de possíveis casamentos que esse meta-símbolo é capaz de cobrir. Um casamento é representado pelo segmento com o qual casa através de dois números, o número da string na qual o segmento começa e o número daquela na qual o mesmo termina.

3. Quais são as restrições do problema?

   Cada meta-símbolo impõe determinadas restrições sobre o texto de entrada. Por exemplo, um meta-símbolo de casamento unitário impõe o seguinte: na posição em que ele se encontra dentro do padrão, deve ser possível casar o meta-símbolo com exatamente um ítem léxico.

A abordagem algorítmica para o problema inicia com a determinação do domínio das variáveis, ou seja, para cada meta-símbolo, quais são os segmentos que podem ser casados. O importante nesse passo é determinar o menor subconjunto possível de valores que podem ser atribuídos dentro de um custo computacional pouco elevado. No presente trabalho, o padrão como um todo é responsável por manter o conjunto de meta-símbolos e por mandar mensagens para que os meta-símbolos se inicializem dados alguns parâmetros básicos, tal como o texto de entrada e o contexto no qual o meta-símbolo se encontra dentro do padrão.

O passo seguinte consiste em determinar quais valores cada variável deve receber tal que todas as restrições sejam satisfeitas. A principal heurística é instanciar primeiro aquelas variáveis cujo domínio é menor, ou seja, as variáveis mais restritas. Cada vez que uma variável é instanciada, todas as variáveis não instanciadas têm seu domínio reduzido, eliminando valores que a variável não pode assumir em função da instanciação atual. Os próprios meta-símbolos são responsáveis, como objetos, por reduzirem seu domínio de acordo com o novo valor. Se uma instanciação fizer com que alguma variável tenha domínio vazio, isso indica que a mesma viola alguma restrição e, consequentemente, a instanciação é desfeita e o próxima é considerada.

O procedimento termina ou quando todas as restrições forem satisfeitas (houve casamento) ou quando o algoritmo determinar que não existe um conjunto de valores que satisfaz todas as restrições ao mesmo tempo (o texto de entrada não casa com o padrão).

É interessante observar como o processo acima é genérico: sempre que for necessário adicionar uma nova restrição para um meta-símbolo, basta que os objetos que representam os meta-símbolos sejam capazes de considerar uma nova restrição, uma vez que são os componentes encarregados tanto de inicializar o domínio como de restringí-lo em função de novas instanciações.

4 Resultados Obtidos

Entre os resultados obtidos, o primeiro é a arquitetura de uma ferramenta de autoria para sistemas tutores de programação para computadores. Dois aspectos fundamentais no sistema de autoria estão consolidados no trabalho até então desenvolvido: a linguagem de especificação da solução (e um compilador correspondente) e a linguagem de especificação de padrões (LAPIDAE).

Com relação ao interpretador tutorial, o resultado foi a determinação de um modelo para as operações utilizadas no processo de diagnóstico da solução proposta pelo aluno. Esse modelo servirá como parte fundamental no desenvolvimento da arquitetura geral do interpretador.

O resultado mais relevante e substancial obtido até a elaboração desse artigo foi a definição e implementação da linguagem de especificação de erros e acertos, LAPIDAE. No sistema em desenvolvimento a LAPIDAE foi construída para ser utilizada pelo mecanismo de detecção de erros ou acertos no programa do aluno, possivelmente melhorando a qualidade do feedback correspondente à descrição de problemas (ou virtudes) na solução do aluno.

Do ponto de vista da flexibilidade e da capacidade de reconhecimento de acerto do aprendiz, a arquitetura proposta será apta, potencialmente, a atingir a mesma flexibilidade do LAURA, embora não necessariamente o protótipo seja desenvolvido para atingir toda a cobertura oferecida pela arquitetura apresentada na Seção 3. Quanto à qualidade de resposta, graças ao mecanismo de transformações por padrão de erro, pode-se esperar a priori um quadro intermediário entre a qualidade de LAURA e dos DAEs existentes.

Da forma como foi concebida a linguagem LAPIDAE, o autor não tem suporte direto para especificação dos objetivos (ou da intenção) do programa-solução proposto. Mas a intenção do programa muitas vezes é essencial para avaliar a lógica de um programa e fazer comentários pertinentes sobre uma solução incorreta, em particular pela capacidade de considerar os erros não sintáticos à luz de um contexto específico[Wenger 1987].

5 Considerações finais

Nesse trabalho apresentamos a problemática de ensinar programação, bem como várias das abordagens já aplicadas para tratar de algumas das questões envolvidas e suas respectivas limitações. Em seguida apresentamos uma arquitetura e alguns dos aspectos consolidados, em particular nas definições de linguagens e das ferramentas que estabelecem a ponte entre o sistema de autoria e o interpretador tutorial. Muitos dos aspectos estão no estágio de planejamento, mas algumas das vantagens e limitações são bastante claras, estando no primeiro a qualidade e flexibilidade do feedback, e nas limitações à qualidade do feedback quando comparado com ferramentas de análise de especificações. Um ponto não levantado, mas que vale a pena considerar, é que claramente uma abordagem que utiliza tranformações tão complexas está sujeita a problemas de complexidade exponencial e até mesmo pior. Ainda deverão ser definidas estratégias para garantir que o sistema responda em um tempo viável.

Nos trabalhos futuros, em longo prazo, é possível melhorar a arquitetura de ambas as ferramentas de autoria e de interpretação. Na autoria, um modelo que estenda a estruturação das classes de erros/acertos e a elaboração de um modelo do aluno (o qual pode ser construído em função da LAPIDAE) são alguns dos objetivos mais próximos. Dentro da ferramenta de interpretação ainda haverão muitas características a serem incorporadas mesmo depois da consolidação do projeto atual. Como mencionado, a ferramenta está atualmente restrita à avaliação simbólica de programas. O sequenciamento do material e uma arquitetura para anotações no modelo do aluno constituiriam alguns dos aspectos essenciais e aproximariam a ferramenta do que deveria ser um tutor inteligente.

Referências

[Adam and Laurent 1980] Adam, A. and Laurent, J. (1980). A system to debug student programs. *Artificial Intelligence*, (15):75–122.

[Aho et al. 1986] Aho, A. V., Sethi, R., and J.D.Ulman (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

[Anderson et al. 1984] Anderson, J. R., Farrel, R., and Sauers, R. (1984). Learning to program in lisp. *Cognitive Science*, (8):87–129.

[Barrett et al. 1985] Barrett, R., Ramsay, A., and Sloman, A., editors (1985). *POP-11: a Practical Language for Artificial Intelligence*. Ellis Horwood.

[Blessing 1997] Blessing, S. B. (1997). A programming by demonstration authoring tool for model-tracing tutors. *International Journal of Artificial Intelligence in Education*, 8:233–261.

[Direne 1997] Direne, A. I. (1997). *Intelligent Training Shells for the Operation of Digital Telephony Stations*, pages 71–78. IOS Press.

[du Boulay and Sothcott 1987] du Boulay, B. and Sothcott, C. (1987). Computer teaching programming: An introductory survey on the field. In Lawler, R. and Yazdani, M., editors, *AI and Education: Learning Environments and Intelligent Tutoring Systems*. Ablex Publishing.

[du Boulay and Sothcott 1988] du Boulay, B. and Sothcott, C. (1988). Intelligent systems for teaching programming. In Ercoli, P. and Lewis, R., editors, *Artificial Intelligence Tools in Education*. Elsevier Science Publishers.

[Gansner and North 2000] Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233.

[Goldstein 1975] Goldstein, I. P. (1975). Summary of MYCROFT: a system for understanding simple picture programs. *Artificial Intelligence*, (6):249–288.

[Hasemer 1983] Hasemer, T. (1983). An empirically-based debugging system for novice programmers. Technical Report 6, Human Cognition Research Laboratory, 10, Open University.

[Johnson 1986] Johnson, W. L., editor (1986). *Intention-based Diagnosis of Novice Programming Errors*. Research Notes in Artificial Intelligence, Morgan Kaufmann.

[Koffman and Blount 1975] Koffman, E. B. and Blount, S. E. (1975). Artificial intelligence and automatic programming in CAI. *Artificial Intelligence*, (6):215–234.

[Lukey 1980] Lukey, F. J. (1980). Understanding and debugging programs. *International Journal Man-Machine Studies*, (12):189–202.

[Munro et al. 1997] Munro, A., Johnson, M. C., Pizzini, Q. A., Surmon, D. S., Towne, D. M., and Wogulis, J. L. (1997). Authoring simulation-centered tutors with RIDES. *International Journal of Artificial Intelligence in Education*, 8:284–316.

[Murray 1999] Murray, T. (1999). Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, 10:98–129.

[Reiser et al. 1985] Reiser, B. J., Anderson, J. R., and Farrell, R. G. (1985). Dynamic student modeling in an intelligent tutor for LISP programming. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence Conference*, pages 8–14, Los Angeles.

[Shapiro 1983] Shapiro, E. Y. (1983). *Algorithmic Program Debugging*. MIT Press.

[Wenger 1987] Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge*. Morgan Kauffmann.