

MODIFICAÇÃO DE COMANDOS COMO INSTRUMENTO DE COOPERATIVIDADE

Antonio L. Furtado^{1,2}, Andrea S. Hemerly² e Marco A. Casanova²

¹Departamento de Informática
Pontifícia Universidade Católica do RJ
R. Marquês de S. Vicente, 225
22.453, Rio de Janeiro, RJ - Brasil

²Centro Científico Rio
IBM Brasil
Caixa Postal 4624
20.001, Rio de Janeiro, RJ - Brasil

SUMÁRIO

Um algoritmo para atingir comportamento cooperativo através da modificação de consultas e atualizações é descrito. Cooperatividade neste caso significa essencialmente contribuir para a satisfação dos objetivos e planos do usuário. O algoritmo é governado por regras e opera em um contexto bem mais rico que o de bancos de dados convencionais. O algoritmo foi desenvolvido e implementado como parte de um projeto mais abrangente, que investiga métodos baseados em conhecimento para criar sistemas de informação cooperativos.

1. INTRODUÇÃO

Descrevemos neste artigo um algoritmo para atingir comportamento cooperativo através da modificação de comandos - consultas ou atualizações - submetidos a um banco de dados. Um sistema de informação exhibe comportamento cooperativo [AP,BJ,CCL,We] na medida em que interage com os usuários de modo a:

- 1) contribuir para a realização dos objetivos e planos dos usuários;
- 2) manter a compreensão do sistema por parte dos usuários em harmonia com a definição e o conteúdo do sistema, evitando mal-entendidos e contribuindo para sua utilização plena e ao mesmo tempo eficiente;
- 3) preservar as restrições de integridade e de autorização estabelecidas.

Para alcançar um comportamento cooperativo, o algoritmo que propomos não executa um comando literalmente, mas o transforma apropriadamente, guiado por um conjunto de regras de modificação. O algoritmo pode modificar um comando acionando regras antes de o comando ser efetivamente executado, depois de o comando ser executado com sucesso, ou mesmo após uma execução do comando que falha. Além do próprio comando, cada regra tem acesso a informações definidas em um rico *contexto de aplicação de banco de dados*, que envolve bem mais do que os componentes de bancos de dados convencionais.

O algoritmo de modificação de comandos foi desenvolvido e implementado como parte do projeto NICE, cujo objetivo é investigar métodos baseados em conhecimento a fim de reduzir o custo de desenvolver "centrais de atendimento" ("*help desks*") e outras interfaces avançadas de bancos de dados do gênero. Neste artigo nos concentramos no uso de regras para modificar um comando. As referências [CF,FC,TF] descrevem, respectivamente, o projeto NICE o gerador de

planos e escalonamentos e as estruturas e algoritmos para bancos de dados temporais.

A noção de modificação de comandos em sistemas de gerência de bancos de dados foi primeiro proposta em [St] para garantir restrições de integridade. O processamento cooperativo de consultas é obtido em [CCL] pela definição de um rico modelo conceitual associado a uma técnica de inferência para especializar ou generalizar consultas. Uma abordagem semelhante é seguida em [CD], que também baseia a transformação de consultas em informação adicional fornecida com o esquema, sob a forma de tópicos de interesse comum. Por outro lado, a generalização de consultas mal sucedidas para atingir comportamento cooperativo é investigada em [Mo]. Um sistema de consulta em linguagem natural a bancos de dados que reconhece pressuposições dos usuários sobre o domínio de aplicação é descrito em [Ka]. O algoritmo que propomos pode aproveitar essas abordagens uma vez que as técnicas de inferência de [CCL] e as regras de transformação de [CD] podem ser diretamente traduzidas em regras disparadas antes da execução do comando, se incorporarmos as extensões propostas por esses autores ao nosso modelo conceitual, e as técnicas em [Ka,Mo] podem ser capturadas através de regras acionadas após execuções mal sucedidas.

Ao examinarmos o algoritmo proposto em comparação com cada uma dessas referências, devemos acentuar que ele abrange tanto consultas como atualizações, e atinge comportamento cooperativo por meio do uso coordenado de regras em vários estágios da execução de um comando. Além disto, o tratamento de atualizações levanta muitas questões interessantes, por envolver técnicas de geração de planos para satisfazer as intenções dos usuários (veja [FC]).

O restante do artigo está organizado como se segue. Apresentamos resumidamente na seção 2 como os vários componentes de uma aplicação de bancos de dados são descritos. Mostramos na seção 3 uma aplicação simples de banco de dados juntamente com exemplos de regras para modificação de comandos. Em seguida, delineamos na seção 4 o algoritmo de modificação de comandos. As conclusões estão na seção 5.

2. COMPONENTES DE UMA APLICAÇÃO DE BANCO DE DADOS

A especificação de uma aplicação em nosso protótipo consiste de um contexto de banco de dados (uma descrição do esquema conceitual, os esquemas externos, as operações orientadas para a aplicação, etc...), um conjunto de regras que capturam o comportamento cooperativo específico da aplicação e, opcionalmente, uma interface de usuário escrita em Prolog. O protótipo contém ainda um conjunto de regras adicionais que capturam comportamento cooperativo de modo independente das aplicações, uma interface auto-contida e um algoritmo para modificação de comandos.

Nesta seção indicamos como definir um contexto de aplicação de banco de dados e quais são os tipos de regras com que o protótipo lida.

2.1 O Contexto de Aplicações de Bancos de Dados

O *contexto* de uma aplicação de banco de dados contém:

- 1) o esquema conceitual do banco de dados, os esquemas externos e as restrições de integridade e de autorização da aplicação;
 - 2) o banco de dados factual;
 - 3) operações pré-definidas orientadas para a aplicação;
 - 4) um modelo do domínio da aplicação;
 - 5) modelos dos usuários, incluindo para cada usuário sua compreensão muitas vezes falha de seus esquemas externos;
 - 6) um repertório de padrões de planos típicos, pré-estabelecidos pelos projetistas de sistemas ou registrados a partir de interações com os usuários;
- e, para cada usuário tendo acesso no momento ao sistema:
- 7) demônios para alertar o usuário sobre a ocorrência de condições especiais;
 - 8) um diário registrando comandos do usuário e as respostas fornecidas.

A implementação corrente especifica os componentes do contexto de modo uniforme, através do uso de cláusulas no estilo de Prolog.

O modelo conceitual é basicamente o modelo entidade-relacionamento com hierarquias *is-a* e algumas outras extensões, tais como *papéis* para as classes de entidade participantes de uma classe de relacionamento. O esquema conceitual de um banco de dados é especificado através de cláusulas que definem as classes de entidades e de relacionamentos existentes e a estrutura das hierarquias *is-a*. Um esquema externo pode em geral ser definido como um procedimento Prolog.

Um *fato* indica a existência de uma instância de entidade ou de relacionamento, ou indica o valor de um atributo para uma instância. Cada fato tem um selo temporal associado. Um *banco de dados* de um esquema conceitual é um conjunto de fatos, possivelmente com selos temporais diferentes.

As restrições de integridade são declaradas diretamente através de cláusulas que indicam conjunções de fatos que não são válidas.

No espírito de tipos de dados abstratos, as operações permitidas sobre um banco de dados são pré-definidas. Cada operação O orientada para a aplicação é também especificada por um conjunto de cláusulas. Estas cláusulas indicam que fatos são adicionados ou removidos por O (i.e., os efeitos de O), as pré-condições para a execução de O , em termos de fatos que devem ou não ser válidos e, opcionalmente, limites inferiores e superiores de tempo para a execução de O .

Uma *instância* O' de uma operação O é um conjunto de cláusulas obtidas pela substituição uniforme das variáveis ocorrendo nas cláusulas que especificam O por termos livres de variáveis. Frisamos que as cláusulas que formam uma instância de operação não contêm variáveis livres.

Por exemplo, as três cláusulas seguintes definem uma operação para matricular um estudante S em um curso C (tanto S como C são variáveis; a sintaxe usada abaixo é uma ligeira simplificação da aceita pela implementação atual):

```
precond(matricula(S,C),estudante(S)).  
precond(matricula(S,C),curso(C)).  
added(toma[estudante(S),curso(C)],matricula(S,C)).
```

As duas primeiras cláusulas dizem que a operação é executada somente se há fatos no banco de dados indicando que S e C são realmente definidos como um estudante e um curso, enquanto a última cláusula diz que, após executar `matricula(S,C)`, o fato `toma[estudante(S),curso(C)]` deve ser adicionado ao banco de dados.

O conjunto de cláusulas a seguir define então uma instância de `matricula` (tanto `s1` como `c3` denotam constantes):

```
precond(matricula(s1,c3),estudante(s1)).  
precond(matricula(s1,c3),curso(c3)).  
added(toma[estudante(s1),curso(c3)],matricula(s1,c3)).
```

Finalmente, observamos que a implementação corrente ainda não trabalha com modelos de usuários nem planos típicos e cobre um modelo do domínio da aplicação apenas através da definição de regras específicas, como discutido em mais detalhe na seção 3. Para cada usuário, o sistema registra o diário da sessão e os demônios criados durante o processamento de seus comandos, novamente em forma de cláusulas. No contexto de processamento cooperativo de comandos, usamos demônios tipicamente para avisar um usuário de uma situação especial, sendo eles criados automaticamente.

2.2 Regras de Modificação de Comandos e Regras de Apresentação

Identificamos duas classes gerais de regras: as *regras de modificação de comandos* (*RM-regras*), que indicam as transformações que os comandos podem sofrer, e as *regras de apresentação*, que definem como a informação obtida em resposta a um comando deve ser editada. Ortogonalmente, distinguimos entre *regras independentes de domínio*, motivadas pelo modelo de banco de dados adotado e *regras dependentes de domínio*, que são específicas do domínio de aplicação em causa.

O protótipo implementa várias regras independentes de domínio. Em lugar de enumerá-las aqui, propomos os seguintes comentários gerais (veja também a seção 3). A possibilidade de definir um conjunto útil de regras independentes de domínio decorre diretamente do modelo conceitual adotado. De fato, conforme argumentado em [CCL,CD], o processamento cooperativo de consultas pode ser obtido definindo um modelo conceitual rico e introduzindo regras independentes de domínio que capturam os padrões de inferência que o modelo permite. Os esforços do projetista da aplicação de bancos de dados canalizam-se então para a definição de informação adicional que acompanha as estruturas conceituais. Pode-se também introduzir regras independentes de domínio para lidar com pressuposições dos usuários, isto é, as impressões sobre o domínio que estes tenham. Um sistema cooperativo deve corrigir qualquer falsa pressuposição do usuário, ao invés de simplesmente falhar. Uma discussão mais detalhada sobre pressuposições pode ser encontrada em [Ka]. Finalmente, a generalização de consultas mal sucedidas, investigada em [Mo], é

também uma fonte de regras independentes de domínio. Já as regras dependentes de domínio são de responsabilidade do projetista de bancos de dados. O protótipo atual requer algum conhecimento de Prolog para formular tais regras, mas pretendemos futuramente introduzir uma linguagem especial para minimizar o esforço dos projetistas.

Classificamos ainda as RM-regras conforme o tipo de comando ao qual se aplicam e quanto à fase de execução em que são usadas. Assim, com base no primeiro critério, temos *regras para consultas* ou *regras para atualizações* e, com base no segundo critério, temos *pré-regras*, que são acionadas antes da execução de um comando, *s-pós-regras*, que seguem a execução bem sucedida de um comando, e *f-pós-regras*, que seguem a execução fracassada de um comando.

Podemos usar pré-regras para corrigir um comando, ou simplesmente para traduzi-lo de uma sintaxe externa "amistosa" para a sintaxe interna de processamento. Podemos também usar pré-regras para complementar um comando, isto é, para suprir informação adicional, em caso de consulta ou atualização, e para executar atualizações adicionais, apenas no caso de atualizações. Já que a necessidade de complementar um comando pode às vezes ser detectada somente após sua execução, temos a alternativa de definir s-pós-regras que geram e executam novos comandos.

Além disto, quando a execução de um comando falha, podemos definir f-pós-regras que geram comandos alternativos para atingir, tanto quanto possível, os mesmos propósitos que o comando original. Podemos também criar uma f-pós-regra que oferece ao usuário a possibilidade de criar um demônio que o alertará quando os obstáculos à execução do comando original não mais existirem. Um obstáculo removível é, por exemplo, uma pré-condição que não é válida quando o comando é submetido, mas que poderá talvez o ser no futuro. Finalmente, quando os obstáculos são de natureza persistente, podemos definir uma f-pós-regra que pelo menos explique porque o comando falhou.

Em resumo, o algoritmo de modificação de comandos primeiro aplica sucessivamente todas as pré-regras possíveis ao comando dado como entrada, produzindo um novo comando. Então, executa o comando modificado contra o banco de dados. Depois de uma execução bem sucedida, pode executar novas ações, como resultado do processamento cumulativo de s-pós-regras. No caso de uma execução mal sucedida, pode aplicar f-pós-regras, uma de cada vez, para oferecer alternativas ao comando. Finalmente, se tudo falhar, tenta f-pós-regras cumulativamente para compensar a falha. O algoritmo usa também regras de apresentação para editar mensagens geradas por RM-regras.

3. EXEMPLOS DE COMANDOS E RESPOSTAS

3.1. Exemplo de uma Aplicação de Banco de Dados

Considere o banco de dados acadêmico de um departamento universitário cujos fatos relevantes identificam alunos, indicam os cursos existentes com seus números de créditos, definem quais alunos estão matriculados em quais cursos com qual nota,

e apontam os cursos de laboratório com seus pré-requisitos de admissão. Os cursos de laboratório são tratados como uma especialização de cursos e, portanto, herdam número de créditos como atributo.

Os exemplos referem-se às seguintes operações: *oferecer* para criar um curso; *matricular* para matricular um aluno em um curso; e *transferir* para transferir um aluno de um curso para outro. A pré-condição para as duas últimas operações exige que o aluno exista e que o curso em que o aluno irá se matricular seja oferecido. Além disso, a estratégia de execução para instâncias de operações (ver seção 4.1) rejeita a matrícula de um estudante no mesmo curso mais de uma vez e a transferência de um aluno de um curso para o mesmo curso.

Os exemplos também supõem que tenha sido declarada uma restrição estabelecendo que nenhum aluno deve estar matriculado em C1 e C2 ao mesmo tempo. Finalmente, os usuários da aplicação são o chefe do departamento, professores e alunos.

Suponha ainda que o banco de dados contenha os seguintes fatos:

- S1 é um aluno;
- os cursos C1 e C2 são oferecidos com 2 créditos;
- o aluno S1 toma o curso C2.

3.2. Exemplos do uso de RM-regras e de Regras de Apresentação

Esta seção exemplifica o modo pelo qual o algoritmo de modificação de comandos (ou, abreviadamente, AMC) usa regras para modificar comandos, em diferentes etapas do processamento de um comando. Os exemplos incluem uma consulta ou uma atualização (identificadas por C e A, respectivamente), seguida de uma descrição das regras envolvidas, onde cada regra é classificada quanto à fase em que se aplica e quanto ao fato de depender ou não do domínio.

O primeiro exemplo envolve f-pós-regras. A pré-regra para consultas corrige um comando baseada unicamente em informação do esquema. A f-pós-regra para consultas corrige um tipo de pressuposição falsa.

Exemplo 1:

C1 - "O curso C4 tem requisito de admissão AC4?"

O AMC aplica primeiro uma pré-regra para consultas para corrigir a consulta para "O laboratório C4 tem requisito de admissão AC4?". Esta regra reconhece o engano comum de referir-se a uma classe de entidades mais geral quando se trata de atributo pertencente a uma classe mais especializada. A consulta modificada falhará para o banco de dados exemplo. Note que o usuário assumiu neste caso que o laboratório C4 é oferecido, o que não ocorre no caso. O AMC então aplica uma f-pós-regra para consultas, reconhecendo a pressuposição falsa do usuário sobre a existência de C4, e replica "Não existe o laboratório C4", o que é mais cooperativo que simplesmente retornar FAIL.

A regra usada para verificar a informação de esquema contida na consulta é:

regra 1 - pré-regra para consultas independente de domínio

Se uma consulta referencia indevidamente um atributo A como pertencente a uma classe de entidades E, quando de fato A pertence a uma classe de entidades F tal que F *is-a* E, a consulta é corrigida substituindo-se E por F e uma mensagem lembrando ao usuário a existência da relação *is-a* é emitida. Se tal correção não é possível, é emitida uma mensagem para comunicar ao usuário que um erro ocorreu: a classe de entidades E não existe; o atributo A pertence a alguma outra classe de entidades E'; ou o atributo A não existe.

O repertório do protótipo também inclui uma regra semelhante para classes de relacionamentos. Inclui ainda uma outra regra, acionada quando um comando de atualização é executado, que verifica se a operação referida realmente existe e se os parâmetros da operação estão corretos.

A regra usada para corrigir o tipo de pressuposição falsa acima é:

regra 2 - f-pós-regra para consultas independente de domínio

Se uma instância de entidade não está presente, mesmo em um relacionamento, então é emitida uma mensagem relatando o fato.

Além disso, a implementação corrente inclui regras que explicam porque uma consulta envolvendo um relacionamento falha. Elas primeiro testam se a consulta falha porque alguma das entidades mencionadas não está presente no banco de dados, e em seguida verificam se as entidades existem mas não estão relacionadas.

O exemplo seguinte usa dois tipos de f-pós-regras: uma para oferecer uma alternativa a um comando mal sucedido e outra para compensar a falha. As regras envolvem pré-condições e restrições de integridade. Além de garantir as restrições de integridade e de autorização, como seria de se esperar, o sistema processa uma atualização cooperativamente, fornecendo explicações, monitorando gatilhos ou oferecendo operações alternativas. Neste último caso, o sistema lança mão do gerador de planos, descrito em [FC].

Exemplo 2:

A1 - "Matricule S1 em C1."

Supondo que nenhuma pré-regra se aplique, o AMC tenta executar a atualização em sua forma original. Entretanto, este comando de atualização falha devido à restrição de integridade que proíbe um aluno cursar C1 e C2 simultaneamente e ao fato de S1 já cursar C2. Assim, o AMC aplica a f-pós-regra para atualizações, onde uma alternativa encontrada pelo gerador de planos consiste em transferir S1 de C2 para C1. O usuário é solicitado a confirmar se concorda ou não com a execução da alternativa. Caso o usuário discorde, o AMC aplica uma f-pós-regra que explica porque a atualização falhou.

As regras aplicadas neste caso são:

regra 3 - f-pós-regra para atualizações independente de domínio

Se a aplicação de uma instância de operação O falha, o gerador de planos tenta achar uma instância alternativa de operação que realize pelo menos todos os efeitos de O.

regra 4 - f-pós-regra para atualizações independente de domínio

Se uma atualização falha porque o estado resultante viola uma restrição de integridade, é emitida uma mensagem explicando a razão da falha. Mas, se a atualização falha porque uma pré-condição ainda não é válida, poderá ser criado um demônio, a ser disparado quando a pré-condição se torna verdadeira, com a finalidade de orientar o usuário a tentar novamente a atualização (o demônio se auto-destruirá após a ativação).

Outra estratégia relevante para atingir comportamento cooperativo consiste em considerar o foco de interesse dos comandos. No próximo exemplo, o algoritmo usa uma f-pós-regra para consultas que tenta capturar o foco dos comandos do usuário durante uma sessão.

Exemplo 3:

C2 - "O curso C5 está sendo oferecido com 2 créditos?"

Supondo novamente que nenhuma pré-regra seja aplicável, o AMC tenta executar o comando em sua forma original. Esta consulta falha para o banco de dados exemplo. Entretanto, suponha que o diário da sessão registre que o usuário já havia submetido outras consultas perguntando por cursos com 2 créditos. Isto indica que o usuário está provavelmente tentando localizar algum curso com 2 créditos. Então, o AMC aplica uma f-pós-regra para atualizações, que gera a consulta alternativa "Ache algum curso com 2 créditos". A nova consulta tem sucesso e C1 é retornado como um curso com 2 créditos.

A regra aplicada neste caso é:

regra 5 - f-pós-regra para consultas independente de domínio

Se falha uma consulta C que pergunta se uma dada instância de entidade tem um dado valor para um atributo, e o diário da sessão indica que uma consulta semelhante C' havia sido formulada antes sobre uma instância diferente, tendo C' igualmente falhado, cria-se uma consulta alternativa C" pela generalização de C para procurar alguma instância satisfazendo o requisito. O algoritmo de generalização mais específica [WM] é usado para selecionar que elementos de C serão transformados em variáveis.

As regras dependentes de domínio podem ou não ser específicas a uma certa classe de usuários. A pré-regra para consultas aplicada no próximo exemplo é específica a alunos de nosso banco de dados acadêmico. O exemplo também mostra uma regra de apresentação.

Exemplo 4:

C3 - "Os cursos C1 e C2 são oferecidos?"

Quando um aluno pergunta sobre um curso, ele provavelmente está pensando em matricular-se e, portanto, o número de créditos é relevante para sua decisão. Assim, o AMC aplica uma pré-regra para consultas que complementa a consulta para "Os cursos C1 e C2 são oferecidos? com quantos créditos?". A consulta modificada tem sucesso no banco de dados usado como exemplo, sendo a resposta ainda não modificada "O curso C1 é oferecido com 2 créditos e o curso C2 é oferecido com 2 créditos". Contudo, o AMC usa ainda uma regra de apresentação para editar a resposta e mudá-la para "todos com créditos = 2", já que ambos os cursos são de 2 créditos. A resposta se torna mais informativa por frisar que o número de créditos é o mesmo.

As regras usadas são:

regra 6 - pré-regra para consultas dependente de domínio

Se um aluno pergunta se um curso está sendo oferecido, a consulta é complementada para retornar também o número de créditos.

regra 7 - regra de apresentação independente de domínio

Se uma consulta pede valores dos mesmos atributos para várias instâncias de entidades, e obtém sucesso com o valor de um ou mais atributos sendo os mesmos para todas essas instâncias, então a resposta é editada para expressar de maneira concisa que tais valores são coincidentes. O algoritmo de generalização mais específica é empregado para este fim.

Como outro exemplo de regra de apresentação independente de domínio, temos uma regra que usa papéis de entidade em um relacionamento para editar uma resposta em linguagem pseudo-natural.

4. O ALGORITMO DE MODIFICAÇÃO DE COMANDO

Descrevemos nesta seção o algoritmo para modificar e executar um comando. Para facilitar a discussão, apresentaremos primeiro como o algoritmo processa um único comando, em seguida como aplica uma única regra a um comando e, finalmente, que estratégias segue para processar diversas regras contra um comando. Por fim, esquematizamos os passos principais do algoritmo separadamente das estratégias de retrocesso ("backtracking").

4.1 Execução de um único comando

Primeiro introduzimos formalmente a sintaxe dos comandos e depois indicamos como são processados, ignorando por enquanto a questão das modificações.

Uma *expressão de consulta* é uma expressão Prolog envolvendo um ou mais fatos do banco de dados.

Um *comando* é uma expressão da forma:

<tipo de comando>(<expressão>, <informação>)

onde, na implementação corrente:

<tipo de comando> é ou *consulta* ou *atualização*. Comandos do primeiro tipo são naturalmente chamados *consultas*, enquanto os do segundo tipo são chamados *atualizações*;

<expressão> é uma expressão de *consulta*, se o tipo é *consulta*, ou uma instância de operação, se o tipo é *atualização*;

<informação> é uma expressão Prolog.

O primeiro parâmetro é chamado *expressão do comando* e o segundo parâmetro é chamado *parâmetro de informação* do comando.

Os usuários podem submeter comandos diretamente através da interface autônoma que o protótipo oferece, ou interagir com o banco de dados através de uma interface especial. No segundo caso, a interface deve ser definida em Prolog e os comandos incluídos em cláusulas Prolog como qualquer comando extra-lógico de Prolog. Devemos também acrescentar que, na maioria das vezes, o parâmetro de informação de um comando é simplesmente uma variável, cujo valor é exibido após uma execução bem sucedida. O parâmetro tipicamente será uma expressão mais complexa, possivelmente contendo variáveis livres, quando o comando for usado em uma cláusula Prolog que inclua outros termos Prolog onde também ocorram algumas dessas variáveis.

Dada uma expressão Prolog E e uma substituição β , denotamos por $E\beta$ a expressão obtida aplicando β a E .

Ignorando modificações, uma expressão C especificada em uma consulta é executada como uma cláusula-objetivo Prolog contra o banco de dados (considerado como um programa Prolog). Isto significa, a grosso modo, buscar uma substituição β das variáveis livres de C tal que $C\beta$ seja uma consequência lógica do banco de dados. Se a busca tem sucesso, $C\beta$ é retornada ($C\beta$ é então chamada uma *resposta* à expressão da consulta). Senão, FAIL é retornado. Pode-se também repetir a busca para determinar outra substituição, se alguma existir, até que não existam novas substituições. No caso especial de C não possuir variáveis livres, a estratégia consiste simplesmente em testar se C é consequência lógica do banco de dados. Se o for, C é retornada (C é também chamada uma *resposta*). Caso contrário, FAIL é retornado como antes.

A estratégia para processar uma instância de operação O , especificada em uma atualização, segue as idéias apresentadas em [VF]. Se as pré-condições de O não valem para o banco de dados corrente, se algum dos efeitos de O já vale no banco de dados corrente, ou se os efeitos de O violam alguma restrição de integridade, nenhuma alteração se produz e FAIL é retornado. Senão, os fatos especificados nas cláusulas de O são adicionados ou removidos, conforme o caso, e todos os demais fatos permanecem inalterados (isto resolve o assim chamado problema do entorno ("frame") [Ko] como seria de se esperar). Por compatibilidade com o processamento de expressões de consulta, O é retornada (O é também chamada uma *resposta*).

4.2 Processamento de uma única RM-regra contra um comando

Introduziremos primeiro a sintaxe formal das regras de modificação de comandos e, em seguida, explicaremos como aplicar uma regra a um comando. Repetiremos depois a discussão para regras de apresentação.

O formato de uma regra de modificação de comando (RM-regra) é:

```
<tipo-regra>(<padrao-de-entrada>,<padrao-de-saida>,<acao>,<mensagem>)  
  <- <condicao>
```

onde, na implementação corrente:

<tipo-regra>	é o <i>tipo</i> da regra, tomado do conjunto {pre_consulta, s_pos_consulta, f_pos_consulta, pre_atualizacao, s_pos_atualizacao, f_pos_atualizacao};
<padrao-de-entrada>	é o <i>padrão de entrada</i> da regra, um termo Prolog que identifica uma classe de expressões de comando;
<padrao-de-saida>	é o <i>padrão de saída</i> da regra, um termo Prolog que resultará na expressão de comando modificada;
<acao>	é a <i>ação</i> da regra, um procedimento Prolog que, agindo sobre <padrao-de-saida> e <mensagem>, cria a expressão de comando modificada e uma mensagem;
<mensagem>	é a <i>mensagem</i> da regra, um termo Prolog cuja avaliação resulta em uma cadeia de caracteres;
<condicao>	é a <i>condição</i> da regra, uma expressão Prolog opcional que deve ser satisfeita para que a regra se aplique.

Notamos que o padrão de saída e a ação podem ter o valor "true", a mensagem pode consistir da cadeia nula e a condição pode estar ausente. Em qualquer desses casos, dizemos que o componente é *trivial*.

Seja r uma RM-regra com o formato acima e seja E uma expressão de consulta ou uma instância de operação. A aplicação de r a E produz uma nova expressão E'' através dos passos seguintes:

- 1) *Teste se a regra se aplica* - Primeiro tente unificar E com <padrao-de-entrada> e teste se <condicao> vale. Se ambos tiverem sucesso, a regra é aplicável a E . Sejam A' , E' e M' as expressões obtidas aplicando as substituições de variáveis resultantes a <acao>, <padrao-de-saida> e <mensagem>.
- 2) *Execute a ação* - Se A' não é trivial, execute-a e sejam E'' e M'' as expressões resultantes da aplicação das substituições assim obtidas a E' e M' . Senão, tome estas expressões como os próprios E' e M' .
- 3) *Gere a saída* - Retorne a nova expressão E'' e, se M'' não for trivial, exiba-a, talvez editada por uma regra de apresentação (conforme discutido abaixo).

A forma de uma regra de apresentação por sua vez é:

```
template(<informacao>,<apresentacao>) <- <condicao>
```

onde, na implementação atual:

<informacao>	é o <i>parâmetro de informação</i> da regra, uma expressão Prolog;
--------------	--

- <apresentacao> é a *apresentação* da regra, uma expressão Prolog que se transforma em uma cadeia de caracteres ao ser exibida e que tipicamente traduz o parâmetro de informação em uma sentença de linguagem pseudo-natural;
- <condicao> é a *condição* da regra, uma expressão Prolog opcional que deve ser satisfeita para que a regra se aplique.

A regras de apresentação são acionadas manualmente usando o comando `pwrite`, cujo único argumento é uma expressão Prolog, ou automaticamente durante o processamento das RM-regras. A execução de `pwrite(A)` se processa da seguinte forma:

- 1) pesquise uma regra de apresentação cujo parâmetro de informação unifica com A e cuja condição, se presente, tenha sucesso;
- 2) se alguma regra existir, exiba seu parâmetro de apresentação, depois de aplicar todas as substituições obtidas do passo anterior;
- 3) senão exiba o argumento A na sua forma original.

O uso de uma regra de apresentação durante o processamento de uma RM-regra ocorre de forma semelhante.

4.3 Estratégias para processar várias regras contra um comando

Dada uma expressão de comando E, há três estratégias básicas para processar um conjunto S de RM-regras contra E:

- *sucessiva* - a expressão de comando E é transformada na expressão final E_n através de uma seqüência de transformações E, E_1, E_2, \dots, E_n . Cada transformação é obtida aplicando-se uma regra. Uma mesma regra pode ser aplicada diversas vezes. O processo termina quando alguma expressão é obtida à qual nenhuma regra em S se aplica.
- *cumulativa* - todas as regras em S são tentadas contra E e o resultado é o conjunto contendo todas as expressões resultantes da aplicação das regras a E.
- *alternativa* - tenta-se aplicar a E, em separado, cada regra em S. Em cada tentativa, apenas uma transformação bem sucedida é considerada.

4.4 O algoritmo principal

O algoritmo de modificação de comandos é basicamente o mesmo para consultas e atualizações. A diferença, que é tratada na próxima seção, reside em como o retrocesso se passa em cada caso.

Seja R o comando e suponha que a sua expressão seja E e que o seu parâmetro de informação seja I. O algoritmo processa R como uma cláusula-objetivo Prolog, isto é, retorna FAIL ou retorna R, com I substituído por um termo via unificação (E não é afetado, mesmo se contiver variáveis livres). Os passos principais são:

1) Correção e complementação do comando original:

Processe pré-regras contra a expressão original E, usando a estratégia de processamento sucessivo, para obter a expressão modificada E' (que permanece a própria expressão E, se nenhuma pré-regra se aplica a E). O

acionamento de uma pré-regra pode determinar que o processamento do comando deve falhar neste ponto.

- 2) *Execução da expressão (possivelmente) modificada:*
 - a) Execute a expressão modificada E' , como explicado na seção anterior.
 - b) Se a execução for bem sucedida, retorne a resposta E'' para E' , e prossiga para o passo 3.
 - c) Senão, prossiga para o passo 4.
- 3) *Complementação da resposta:*
 - a) Aplique s-pós-regras a E'' , usando a estratégia de processamento cumulativa. Cada aplicação de uma s-pós-regra a E'' que for bem sucedida retorna uma nova expressão. Seja E_1, E_2, \dots, E_m a seqüência de expressões assim obtidas.
 - b) Execute cada E_i , como explicado na seção anterior, retornando a resposta E''_i para E_i ou FAIL. Seja $E''_{j_1}, E''_{j_2}, \dots, E''_{j_n}$ a seqüência de soluções assim obtida (ou seja, ignore as expressões E_i que falharam).
 - c) Unifique o parâmetro de informação I do comando original R com $E'' \& E''_{j_1} \& E''_{j_2} \& \dots \& E''_{j_n}$ ou simplesmente com E'' , se a seqüência for vazia.
 - d) Se a unificação falhar, prossiga para o passo 4.
 - e) Senão pare e retorne o comando original R , com I modificado pela unificação acima.
- 4) *Busca de alternativas:*
 - a) Aplique f-pós-regras com um padrão de saída não-trivial a E' , usando a estratégia de processamento alternativa.
 - b) Execute cada expressão assim obtida, sem acionar o processo de modificação de comandos recursivamente, até que uma tenha sucesso e retorne uma resposta E'' que unifique com o parâmetro de informação I do comando original R , ou que todas tenham sido tentadas. Para atualizações, cada alternativa não é diretamente executada, mas apenas exibida ao usuário, que deverá então decidir se ela deve ou não ser executada.
 - c) Se o passo anterior falhar, prossiga para o passo 5.
 - d) Senão, pare e retorne o comando original R , com I unificado com E'' .
- 5) *Compensação de falhas:*
 - a) aplique f-pós-regras com um padrão de saída trivial a E' , usando a estratégia de processamento cumulativa. Tais regras podem exibir mensagens, criar demônios ou produzir outros efeitos colaterais.
 - b) Pare e retorne FAIL.

4.5 Estratégias de Retrocesso

O algoritmo usa estratégias de retrocesso distintas para consultas e atualizações. A estratégia é muito liberal para consultas já que elas não produzem efeitos nos dados armazenados. Logo, começamos discutindo como se dá o processo de retrocesso para os vários passos do algoritmo com respeito a consultas.

Não há retrocesso para o passo de correção/complementação do comando original, dado que apenas uma seqüência de transformações deve ser formada. Esta decisão simplifica a verificação de que o processo de transformação termina e evita uma explosão combinatorial que, de outra forma, ocorreria se se considerasse todas as possíveis seqüências de transformações. Por outro lado, esta decisão não assegura que a expressão modificada seja a melhor possível.

Pode-se forçar retrocesso sobre o passo de execução se existir mais de uma resposta para a consulta (modificada). Para cada resposta, o passo de complementação de resposta (que por si só não sofre retrocesso, já que a estratégia cumulativa o torna desnecessário) naturalmente é repetido.

Da mesma forma, pode-se forçar retrocesso sobre o passo de busca de alternativas a fim de obter todas as possíveis alternativas. Para maior liberdade, estabelecemos ainda que o passo de tentativa de alternativa pode ser alcançado forçando-se retrocesso além da última execução bem sucedida do passo de execução e complementação de respostas. Além disto, o passo de compensação de falha pode também ser alcançado forçando-se retrocesso além da última execução bem sucedida do passo de busca de alternativas.

Para atualizações, o ponto essencial reside em que apenas uma execução bem sucedida de uma instância de operação pode ter lugar quando o comando de atualização é chamado. Logo, se o passo de execução tem sucesso para a instância (modificada) da operação, o passo de complementação da resposta é executado e então o algoritmo pára. Senão, o passo de busca de alternativas é executado. Além disso, se uma alternativa é aceita, o algoritmo a executa e pára. Se nenhuma alternativa existe ou se todas são rejeitadas, o passo de compensação de falha é executado uma só vez.

5. CONCLUSÃO

Uma primeira versão do protótipo NICE, cujo algoritmo central para modificação de comandos foi descrito neste artigo, está operacional.

Nossos esforços concentram-se agora em estudar os seguintes pontos. Primeiro, pretendemos estender o repertório de regras de modificação independentes de domínio por serem estas as de maior impacto. Estamos também desenvolvendo uma teoria para modelagem de usuários e um método para projetar e implementar tais modelos. Além disso, devemos estudar como customizar o comportamento cooperativo para classes de usuários e para usuários individuais [KW,NS]. Finalmente, investigaremos como incorporar a idéia de planos típicos.

Com o prosseguimento de nossa pesquisa, esperamos analisar exemplos mais complexos, que explorem em maior profundidade aspectos interessantes do comportamento cooperativo considerando a noção de tempo e que sejam tratáveis em nosso banco de dados temporal.

REFERÊNCIAS BIBLIOGRÁFICAS

- [AP] J. F. Allen e C. R. Perrault - "Analyzing intentions in utterances" - Artificial Intelligence 15, 3 (1980) 143-178.
- [BJ] L. Bolc e M. Jarke (eds.) - "Cooperative Interfaces to Informação Systems" - Springer-Verlag (1986).
- [CCL] W. Chu, Q. Chen e R-C. Lee, "Cooperative Query Answering via Type Abstracção Hierarchy", Proc. Int. Working Conference on Cooperating Knowledge based Systems, Keele, Inglaterra (1990).
- [CF] M. A. Casanova e A. L. Furtado - "An Informação System Environment based on Plan Generation", Proc. Int. Working Conference on Cooperating Knowledge based Systems, Keele, Inglaterra (1990).
- [CD] F. Cuppens e R. Demolombe - "Cooperative answering: a methodology to provide intelligent access to databases" - Anais da Second International Conference on Expert Database Systems - L. Kerschberg (ed.) - Benjamin/Cummings (1989) 621-643.
- [FC] A. L. Furtado e M. A. Casanova - "Plan e schedule geração over temporal databases" - Anais da 9th International Conference on Entity-Relationship Approach (1990).
- [Ka] S. J. Kaplan - "Cooperative Responses from a Portable Natural Language Query System" - Artificial Intelligence 19, 2 (1982) 165-187.
- [Ko] R. Kowalski - "Logic for problem-solving" - North-Holland Pub. Co. (1979).
- [KW] A. Kobsa e W. Wahlster (eds.) - "User Models in Dialog Systems" - Springer-Verlag (1989).
- [Mo] A. Motro - "Query generalization: a technique for handling query failure" - Anais do First International Workshop on Expert Database Systems (1984) 314-325.
- [NS] E. J. Neuhold e M. Schrefl - "Dynamic derivação of personalized views" - Anais da 14th VLDB Conference (1988) 183-194.
- [St] M. R. Stonebraker - "Implementação of integrity by query modification" - Anais da ACM SIGMOD International Conference on Management of Data (1975).
- [TF] L. Turcherman e A. L. Furtado - "Update-oriented database structures" - Anais of the Second International Conference on Expert Database Systems (1988) 185-203.
- [VF] P. A. S. Veloso e A. L. Furtado - "Towards simpler e yet complete formal specifications" - in "Informação systems: theoretical e formal aspects" - A. Sernadas, J. Bubenko e A. Olive (eds.) - North-Holland Pub. Co. (1985) 175-189.
- [We] B. L. Webber - "Questions, answers e responses: interacting with knowledge base systems" - in "On knowledge base management systems" - M. L. Brodie e J. Mylopoulos (eds.) - Springer (1986).
- [WM] A. Walker, M. McCord, J. F. Sowa e W. G. Wilson - "Knowledge systems e Prolog" - Addison-Wesley Pub. Co. (1987).